A Study on the Acceleration of Arrival Curve Construction and Regular Specification Mining using GPUs

by

Nirmal Joshi Benann Rajendra

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 3 of this thesis has been adopted from the work [7] that was accepted for publication under IEEE Design & Test (Volume: 35, Issue: 4, Aug. 2018). Author of this thesis is one of the co-authors for the accepted paper along with the first author, Carvajal, G., Salem, M., and Fischmeister, S. In this chapter, Carvajal, G., Salem, M. and Fischmeister, S. aided in formalising the idea of the construction of the arrival curves. Carvajal, G. also aided in the developing the intuition and approach for the construction of arrival curves.

All the work presented in this thesis were completed under the supervision of Fischmeister, S. in the Electrical and Computer Engineering department at the University of Waterloo, who contributed with his ideas, reviews,and suggestions.

Abstract

Data analytics is a process of examining datasets using various analytical and statistical techniques. Several tools have been proposed in the literature to extract hidden patterns, gather insights and build mathematical models from large datasets. However, these tools have been known to be computationally demanding as the datasets become larger over time. Two such recently proposed tools are the construction of arrival curves from execution traces and mining specifications in the form of regular expressions from execution traces. Though the architectures in CPUs have extensively improved over the years to execute such computationally intensive tasks, further enhancements have been impeded due to increased heat dissipation. This has resulted in enabling parallel computing through GPUs as a vastly favorable alternative to overcome the computational challenges.

In this thesis, we present an exploratory work on applying GPU computing to the construction of arrival curves and mining specifications in the form of regular expressions as case studies. The novel approaches taken for each of the case studies are first presented followed by the algorithmic breakdown to expose the parallelism involved. Lastly, experiments using commodity GPUs are presented to showcase the significant speedups obtained in comparison to the equivalent non-parallel implementations.

Acknowledgements

I would like to extend my gratitude to everyone who has supported me throughout my masters.

First and foremost, I would like to express my deepest gratitude towards my supervisor, Prof. Sebastian Fischmeister for being an approachable supervisor and giving me all the opportunities to help me grow both on a technical and personal level. Thank you for making time to meet with me even though you always had a busy schedule.

I am also extremely grateful to Dr. Gonzalo Carvajal. Thank you for always being there to offer guidance and support ever since I was a URA. Thank you for going the extra mile to check up on me and making sure I was in the right direction. Also, a special thanks to his students - Andrew, Hans, Jaime, Mario and Pedro.

I would like to also immensely thank Dr. Apurva Narayan for being my guide and support since my first day of masters. Thank you for the pleasure of having the experience to work, research with you and for always being there in person to help me whenever I needed it the most.

I thank my readers, Prof. Hiren Patel and Prof. Paul Ward for their valuable time in reviewing my thesis.

I felt incredibly privileged and humbled to have worked with my friends and colleagues at the Real-time Embedded Software Group especially Adan, Anderson, Anson, Carlos, Giovani, Jack, Mahmoud, Murray, Oleg, Sean and Waleed. I sincerely thank you all for your company and for sharing your knowledge. Being a part of the Embedded Soccer Group was awesome.

Special shoutout and thanks to David YeounJun Park and Tommy Nguyen for always being there and accompanying me as I transitioned from Bachelors to Masters with them. I would like to also thank Andrew Alberts for being a friend and that genius classmate who helped me complete CS 666. To Bharat, Gowtham, George, Hareesh, JJ, Mahesh, Ranga, Sara, Sathyan and Vicky, thank you for the weekend football memories.

My heartfelt thanks to my family especially my parents for their unconditional love, guidance, support and always being there. They are the backbone of my success and achievements.

Last but not least, I would like to thank God for giving me the strength, hope and making all of this possible.

Dedication

This thesis is dedicated to my dad - Mr. Benann Thiruthuvanathan, my mom - Mrs. Carmel Mercy Benann and my brother - James Benann. I will always be grateful to them for supporting me, no matter what decision I make.

Table of Contents

List of Tables

List of Figures

Abbreviations

ACE Asynchronous Compute Engine 13

AMD Advanced Micro Devices, Inc. 12–16, 28, 29, 60, 65

API Application Programming Interface 3, 13

CPU Central Processing Unit 1, 2, 6, 7, 10, 12–16, 27, 29, 30, 56, 64

CU Compute Unit 13, 14, 16, 60, 61

DDR Double Data Rate 65

DFSM Deterministic Finite State Machine 34, 35, 37–41, 43, 45, 47

DPP Data-Parallel Processor 12, 13

DRAM Dynamic Random Access Memory 5

FPGA Field Programmable Gate Array 2, 65

GCN Graphics Core Next 12

GCP Graphics Command Processor 13

GDS Global Data Share 15

GPC Graphics Processor Cluster 5

GPGPU General Purpose GPU 3, 12

GPU Graphics Processing Unit 2–8, 10, 12, 14–16, 19, 27–30, 33, 49, 53, 55–57, 59–61, 64, 65

LDS Local Data Share 13, 16

NC Network Calculus 18

NW Nested Word 33, 36, 39, 43, 45, 47, 48, 52–54

NWA Nested Word Automaton 39, 46

NWAs Nested Word Automata 45

OpenCL Open Computing Language 3–5, 13–16, 27–29, 47, 49, 53, 55, 56, 58–61, 64, 65

OpenMP Open Multi-Processing 65

PE Processing Element 14

PEs Processing Elements 14

RTC Real-Time Calculus 20

sALU scalar Arithmetic Logic Unit 13

sGPR scalar General Purpose Register 13

SIMD Single-Instruction Multiple-Data 3, 6, 8, 10, 13, 14

SIMT Single-Instruction Multiple-Thread 8

SM Streaming Multiprocessor 5, 6, 8–10, 14, 60

SP Streaming Processor 6, 9

TRE Timed Regular Expression 33, 34, 36, 38–40, 43, 45, 47–49, 53, 65

vALU vector Arithmetic Logic Unit 13, 61

vGPR vector General Purpose Register 13

VLIW Very Long Instruction Word 62

Chapter 1

Introduction

Data generated by numerous information sensing systems such as Internet of Things (IoT) devices, mobile applications, sensors, websites and social media are primarily large and unstructured. Applying advanced data analytics techniques based on data mining, machine learning, abstraction of mathematical models to these large datasets can help analyze and build structured datasets. These advanced techniques can be used to build a broad spectrum of reliable tools that can automate the data analysis process for any given set of data. However, these data analysis tools can be computationally demanding as the datasets increase in size.

Examples of two such tools are the construction of arrival curves and mining of arbitrary regular specifications. Arrival curves are well-known abstractions for mathematically modelling the temporal executions in real-time systems. There exists substantial literature that uses arrival curves for the design and analysis of real-time systems [7]. However, a tool for the construction of arrival curves has been ignored or omitted due to the computational challenges involved [7]. Similarly, specification mining is the process of examining execution traces of sophisticated programs to identify patterns of event occurrences. Mined specifications represent formal properties that characterize a program's dynamic behaviour, which can be used for debugging, verification, anomaly detection, among other applications [30, 29]. Recent literature reports multiple techniques and tools for automatic mining of temporal properties expressed in the form of regular expressions (hence, the term regular specification mining), which can be represented using deterministic finite state machines. However, these mining algorithms suffer from the time complexity arising from the inherent formulation of the automaton and the size of execution traces as shown in [30, 29].

For the last 40 years, one of the significant ways to boost the Central Processing Unit

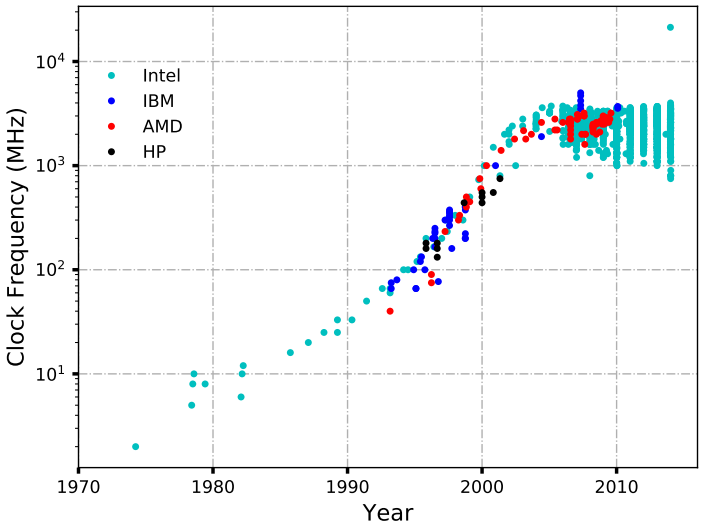


Figure 1.1: CPU Clock Frequency Trend over Time [17]

(CPU) performance for computationally demanding tasks was by achieving gains through clock frequency [45]. Increasing clock frequency means completing tasks faster. However, as shown in Figure 1.1, the trend of clock frequency over time was growing steadily but started to flatten around early 2000. The reason behind the trend is that increasing clock frequency results in increasing the complexity of the CPU hardware which in turn causes heat dissipation in large amounts and high power consumption [45]. This limitation on the CPU clock frequency improvement resulted in turning the spotlight to parallel computing strategies for computationally intensive tasks. Specifically, applying parallel computing through heterogeneous systems involving Graphics Processing Unit (GPU)s appear to be a valuable solution as the commodity parallel hardware are becoming cost-effective [40, 50].

Heterogeneous systems consist of a host and connected device(s). Generally, the host is a CPU. The host delegates or offloads computationally intensive tasks to the device(s) and then collects the results of the tasks from the device(s) yielding in an overall better timing performance relative to a CPU-only execution. The device(s) are generally parallel computing device(s) that execute the tasks in parallel. Examples of such devices can be CPU, GPU, Field Programmable Gate Array (FPGA), and Digital Signal Processor (DSP).

Originally, GPUs were designed to accelerate the graphics processing pipeline. This involved having a fraction of the graphics processing pipeline stages implemented in hardware that were optimized, task-specific and performed fixed functions [34]. Though some of these stages were configurable, they lacked programmability. Eventually, programmability

was added to the graphics pipelines to enable flexibility of certain features in graphics such as texture and lighting [34]. Over time, as the graphical applications increased in complexity, GPUs evolved primarily focusing on the programmable stages of graphics pipeline [34]. Modern GPU architectures contain large arrays of programmable processing units. In other words, older generations of GPUs can be best described as a task-specific pipeline with additions of programmability whereas modern GPUs can be described as a programmable engine with task-specific units [34]. The programmable units of modern GPU use Single-Instruction Multiple-Data (SIMD) programming model [34]. This implies that the GPU can process different data points or different elements in a dataset in parallel using a single instruction. This support of having programmable units with a parallel programming model in GPUs opened opportunities for exploring the use of GPUs for general purpose computations.

Efforts were made by researchers to map the general purpose applications to the existing graphics Application Programming Interface (API) in order to use the GPUs for general purpose computing [50]. As a result, the GPUs used for such computing purposes are known as General Purpose GPU (GPGPU). This mapping to graphics API implies each computation had to be expressed and reduced in terms of pixels even if the computation had no relation to graphics [34]. This obfuscated the use of GPU for general purpose computing thereby causing debugging to be very tedious [34]. Eventually, this limitation was overcome through the development of commercial tools and programming environments such as Open Computing Language (OpenCL) and CUDA enabling direct and non-graphical interface with the programmable units. These tools provide programming environments to exploit an application's parallel computing opportunities by providing the means to specify the parallelism through high-level programming. Hence, such tools provide a high-level programming model that opens avenues to take complete advantage of GPU's powerful architecture while enabling productive implementations of compute-intensive applications such as data analysis tools [34].

To showcase the interest of using heterogeneous systems containing GPUs to improve the performance of data analysis tools, we propose an exploratory study in this thesis on applying accelerators such as GPUs (with OpenCL and CUDA) to data analysis tools like the construction of arrival curves and regular specification mining. The approach taken in this study uses OpenCL GPUs as primary accelerators followed by exploring the results of implementing the same algorithm on CUDA GPUs.

1.1 Contributions

The main contribution of this thesis are:

- Propose two novel algorithms - construct arrival curves and mine specifications in the form of regular expressions from execution traces to show the application of accelerators to tools.
- Discuss the computational approach to formulate both the algorithms. This also includes the discussing the parallelism involved and the acceleration results of using parallel hardware for such algorithms.
- Present the abstractions in the form of optimizations that were considered when applying GPU computing to both case studies. These abstractions can be considered while applying CUDA and OpenCL GPU accelerators to other tools or algorithms.

1.2 Organization of Thesis

The remainder of this thesis advances as follows: We discuss the background required for GPU computing in chapter 2 by going over the architecture and programming model of the GPUs used. Chapters 3 and 4 present the case studies of the construction of the arrival curves and the regular specification mining respectively along with the results of the acceleration. Chapter 5 discusses the common abstractions that were considered as a result of using the two GPU programming models in the case studies. Lastly, Chapter 6 includes concluding remarks along with future work.

Chapter 2

Background

This chapter describes the architecture of GPUs along with the programming models of the tools used for each of them. GPUs used in this study are manufactured by Nvidia and AMD. Hence, the Nvidia GPUs are programmed using CUDA, and the AMD GPUs are programmed using OpenCL.

2.1 Nvidia GPU

This section will present a general overview of a Nvidia GPU architecture. The detailed specifications of a Nvidia GPU depend on the architecture model of the GPU like Pascal and Fermi. An illustration of one of Nvidia's earliest GPU architecture, the Fermi architecture, is shown in Figure 2.1. In order to get the simplified overview of the Nvidia GPU architecture and to understand the programming model described later, the key components in the Nvidia GPU architecture are the following:

- **Graphics Processor Clusters (GPCs)**
- **Streaming Multiprocessors (SMs)**
- **Streaming processors (SPs) (also known as CUDA cores)**

Each Graphics Processor Cluster (GPC) contains a collection of SMs along with the Dynamic Random Access Memory (DRAM) global memory and it is interconnected to the other GPCs. Each Streaming Multiprocessor (SM), in turn, consists of many SPs, thread

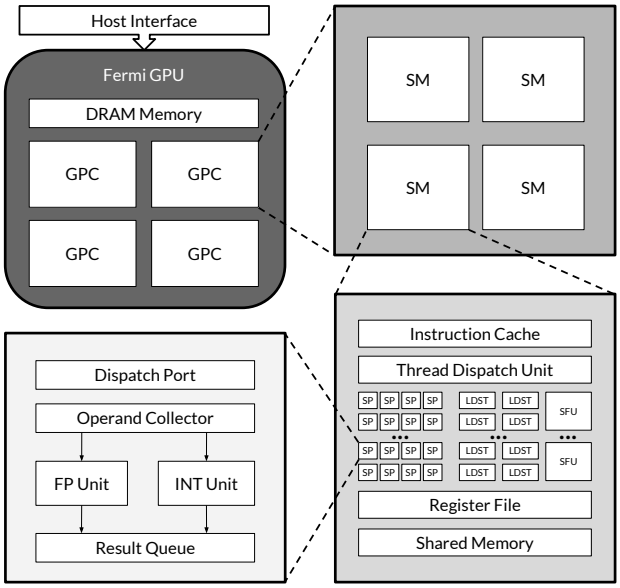


Figure 2.1: Overview of Nvidia's Fermi Architecture

dispatch unit, instruction cache, shared memory, register file, load/store unit (LDST) and special function units (SFU) which perform special hardware instructions such as sine/cosine/exponent operations. The thread dispatch unit dispatches the threads to the Streaming Processor (SP). Usually, the number of threads handled in a SM is 1024. Each SM further divides the number of threads into *wraps*. The SPs are responsible for performing various complex computations. To perform such computations, each SP contains an Integer Unit (INT Unit) and Floating Point Unit (FP Unit). Over time, many architectures such as the new Turing architecture were developed by improving on the Fermi architecture and by adding more of the SPs thereby, increasing the throughput.

2.1.1 Programming in CUDA

The CUDA programming model is one of the many tools available for programming a heterogeneous system. As mentioned before, a heterogeneous system consists of a host which is the CPU and a device which in this case is a Nvidia GPU. The host and the device also contain their own memory - host memory and device memory respectively. The function that gets executed on the GPU is called the kernel. This is the key component for enabling parallelism through SIMD when using tools like CUDA. The kernel enables the programmer to focus on the design and logic of the application algorithm instead of

worrying about the details on thread allocation or deallocation in GPU. In other words, the kernel can be written as a sequential program which is then used by CUDA to schedule the kernel on the GPU threads [9]. The high-level flow of a CUDA program is the following:

1. CPU allocates memory locally for the given data.
2. CPU allocates GPU memory and transfers the data from CPU to GPU.
3. CPU launches the kernels asynchronously on the GPU to execute on the copied data.
4. CPU waits on GPU to finish and then copies the results of the execution to CPU.

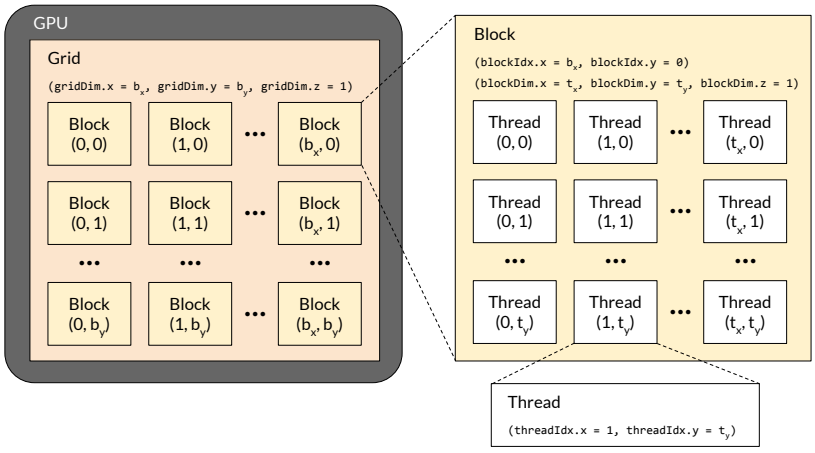


Figure 2.2: CUDA Thread Hierarchy

It is important to note that the control is returned back to the CPU after launching the kernel(s) thereby, enabling the CPU to perform any other additional tasks. Each line or instruction in the kernel is executed by a large number of threads in the GPU. CUDA allows a two-level thread hierarchy that enables the programmer to organize the threads required to execute a kernel. Organizing the number of threads is a critical part to exploiting the parallelism within Nvidia GPU. The thread hierarchy is such that the threads are grouped into blocks and the blocks are further grouped into a grid as shown in Figure 2.2. In other words, a grid is made up of blocks and each block in a grid is made up of threads. The threads inside a block can synchronize amongst themselves and also cooperate among each other using shared memory. However, threads from different blocks cannot interact and cooperate with each other. CUDA offers built-in variables to uniquely identify a thread as listed below:

- `blockIdx` - Holds the block index within a grid.
- `threadIdx` - Holds the thread index within a block.
- `blockDim` - Holds the block dimension measured in terms of threads.
- `gridDim` - Holds the grid dimension measured in terms of blocks.

Each of these variables are of type `uint3`, a built-in CUDA `struct` containing three variables - `x`, `y` and `z`. These can be accessed inside a kernel using the following syntax: `blockIdx.x`, `blockIdx.y`, `blockIdx.z`, `threadIdx.x`, `threadIdx.y`, `threadIdx.z`, `blockDim.x`, `blockDim.y`, `blockDim.z`, `gridDim.x`, `gridDim.y` and `gridDim.z`. This implies that the CUDA organzies the grids and blocks in 3-dimensional. By default, the `x`, `y` and `z` variables of the variables holding the dimension value is set to 1.

In order to launch a kernel with configurations on the number of threads required, the following syntax can be used -

```
kernel_name<<<grid_dim, block_dim>>>(list of arguments)
```

The variable `grid_dim` can be of type `uint3` that indicates the number of blocks per grid along each of the three dimensions of the grid. Similarly, the variable `block_dim` can be of type `uint3` that indicates the number of threads per block along each of the three dimensions of the block. As an example, the following declaration: `uint3 grid_dim(512);` indicates 1-D grid containing 512 blocks. Similarly, this `uint3 block_dim(1024, 1024);` declares a 2-D block containing 1024 threads along each of the 2 dimensions. These variables can then be used to launch the kernels.

After launching a kernel with some grid and block dimensions, the blocks are distributed and scheduled to the SMs for execution. Multiple thread blocks can be assigned to the same SM depending on the resources available in the GPU. However, once a thread block is scheduled in a SM, it remains in that SM until the execution is complete. The scheduling and the distribution of the thread blocks to the SMs is taken care by CUDA.

CUDA uses a Single-Instruction Multiple-Thread (SIMT) model for thread management and execution. The major difference between SIMD and SIMT is that SIMD model executes a single instruction on multiple data points whereas in a SIMT model, multiple threads execute the same instruction independently. This group of threads that execute the same instruction at the same time is called a wrap which is usually made up of 32 threads [9]. Each SM divides the block of threads further into wraps and then, schedules the execution of threads.

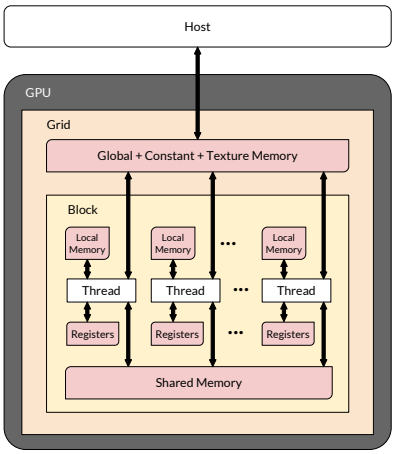


Figure 2.3: Overview of the CUDA Memory Model

2.1.2 CUDA Memory Model

The main types of programmable memory in the CUDA memory model are register file, shared, local, constant, texture and global memory. Overall view of the memory model is shown in Figure 2.3. Each SM contains a register file which acts like private memory for each thread by allowing storage of registers required by the threads. There is effectively negligible wait time on the register file and hence has the fastest memory access [11]. The lifetime of the register variables is shared with the kernel i.e. the variables cannot be accessed once the kernel finishes execution [9]. Shared memory is present in each SM as well and accessible by all the SPs present in the SM. The threads in a thread block cooperate with each other using the shared memory. This can result in race conditions if the threads were to be executed in an undefined order. However, CUDA primitives are available to prevent such race conditions within a thread block [9]. The shared memory is limited in each SM and has higher latency than that of the register file but the lifetime of the shared memory is shared with the thread block.

The local memory is a special view of the global memory used for register variables unable to fit into the allocated register space such as large local structures or arrays and variables that exceed the register limit [9]. Since local memory is in the same physical location as that of the global memory, it has the same bandwidth and latency as that of the global memory. The constant memory is present in the device memory identical to the global memory and contains only read-only data for the kernels. It is declared in the global scope and is visible to all the kernels. Another memory that resides in the device like the global memory is the texture memory which is useful for storing 2D or 3D data. Texture

memory can be accessed only through a read-only cache [9] as the cache performs floating-point interpolation. This facility in Texture memory is advantageous to applications that are designed around 2D or 3D data.

Lastly, the global memory is the largest memory available on the GPU. It has the highest latency and is also most commonly used memory due to its size. The global memory is available to all the SMs at the same time and its lifetime is shared with the application. This can cause undefined program behaviour due to race conditions arising from threads in two different thread blocks accessing the same global memory location. Hence, it is advised to take caution when designing the memory access to the global memory. The CPU in the heterogeneous system transfers the data to the GPU by first allocating memory in the global memory and then moving the data from the CPU memory to the GPU memory.

In addition to these, the Nvidia GPU also contains non-programmable caches: L1 cache, L2 cache, read-only constant and read-only texture [9]. Each SM contains an L1 cache, a read-only constant cache, a read-only texture cache and the L2 cache is shared among all the SMs. The L1 and L2 caches are used to store the data in local and global memory whereas the read-only constant cache and the read-only texture cache is used to store data in the constant and texture memory respectively.

2.1.3 Example

Figure 2.4 shows an example CUDA program that does vector addition in parallel. This program can be stored with the `.cu` file extension. The CUDA compiler driver, `nvcc` that is part of the CUDA toolkit, follows the CUDA compilation trajectory as listed below:

- Separating the device functions (like kernels) from the host code
- Compile the device functions using the Nvidia compiler
- Compile the host code using a C++ compiler
- Link CUDA runtime libraries to support SIMD steps along with GPU memory allocation and data transfer between CPU and GPU

This example starts with the initialization of two vectors of size 1024 each and each containing consecutive numbers starting from 1. After initializing, the necessary GPU memory is allocated using `cudaMalloc` and data is transferred from CPU to GPU using `cudaMemcpy`. The kernel in this example is the `doSum` function and gets executed on the

```
1  #include <cuda_runtime.h>
2
3  #define SIZE 1024
4  #define START 1
5  using namespace std;
6
7  __global__ void doSum(int *a, int *b, int *sum) {
8     unsigned int i = threadIdx.x ;
9     sum[i] = a[i] + b[i];
10 }
11
12 void initVector(vector<int> &v) {
13    int x = START;
14    for (int i = 1; i <= SIZE; ++i) {
15       v.push_back(x++);
16    }
17 }
18
19 int main(int argc, char *argv[]) {
20    vector<int> h_a, h_b;
21    vector<int> h_sum(SIZE, 0);
22    int *d_a, *d_b, *d_sum;
23
24    initVector(h_a);
25    initVector(h_b);
26    cudaMalloc((void**)&d_a, sizeof(int)*SIZE);
27    cudaMalloc((void**)&d_b, sizeof(int)*SIZE);
28    cudaMalloc((void**)&d_sum, sizeof(int)*SIZE);
29    cudaMemcpy(d_a, h_a.data(), SIZE*sizeof(int),
         cudaMemcpyHostToDevice);
30    cudaMemcpy(d_b, h_b.data(), SIZE*sizeof(int),
         cudaMemcpyHostToDevice);
31
32    doSum<<<1, 1024>>>(d_a, d_b, d_sum);
33
34    cudaMemcpy(h_sum.data(), d_sum, SIZE*sizeof(int),
         cudaMemcpyDeviceToHost);
35    cudaFree(d_a);
36    cudaFree(d_b);
37    cudaFree(d_sum);
38
39    return 0;
40 }
```

Figure 2.4: Vector Addition Example in CUDA

GPU. This kernel is launched with a 1-D grid containing a single 1-D thread block made up of 1024 threads. Each thread runs the kernel by first obtaining its thread index in the block using `threadIdx.x` followed by the addition operation at the vector index equal to the thread index. While the addition is done in parallel, CPU is waiting on the `cudaMemcpy` call to copy the results from the GPU to CPU. The results will be copied into the CPU memory once all the threads complete the kernel execution.

2.2 AMD GPU

An alternative to using a Nvidia GPU in a heterogeneous system is an Advanced Micro Devices, Inc. (AMD) GPU. The codename for the microarchitecture used in AMD GPUs launched from 2011 is Graphics Core Next (GCN). This microarchitecture was launched to improve the architecture of its predecessor, TeraScale, especially for GPGPU computations. This section discusses the components of the GCN in a AMD GPU used in a GPGPU setting. Figure 2.5 shows an overview of the GCN architecture.

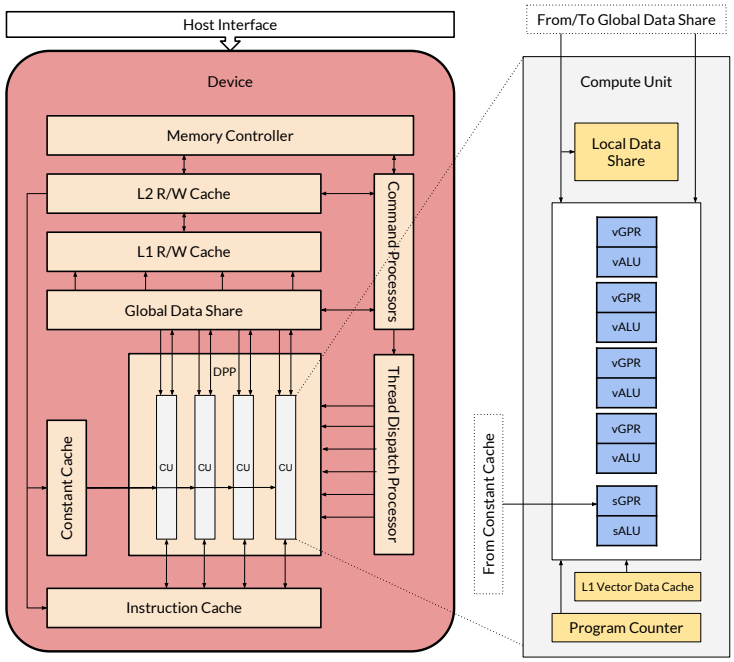


Figure 2.5: Overview of the Graphics Core Next Architecture

The main component that forms the heart of the GCN is the Data-Parallel Processor

(DPP) array [5]. The DPP contains a collection of compute units that operate independent of each other and in a parallel manner on data streams of floating-point or integer type [5]. Each Compute Unit (CU) is the basic computational block in the architecture. A CU is made up of 4 SIMD vector units (stream processors), a program counter, scalar General Purpose Register (sGPR), scalar Arithmetic Logic Unit (sALU) and Local Data Share (LDS) and a L1 vector data cache [3]. Each SIMD vector unit contains a vector General Purpose Register (vGPR) and vector Arithmetic Logic Unit (vALU) capable of executing single or double precision floating point operations simultaneously using 16 threads (work-items) [3]. Hence, 4 SIMD vector units results in total of 64 threads to execute simultaneously which is called a *wavefront* and this term is analogous to a wrap in CUDA. The LDS within each compute unit is used as a register file to synchronize among the different SIMD units and is designed to have low-latency bandwidth [3].

The command processors are responsible for receiving high-level API commands from the driver and mapping them onto the two processing pipelines - Asynchronous Compute Engine (ACE) and Graphics Command Processor (GCP) [3]. The ACE is responsible for parallel compute operations and the GCP is responsible for graphics operations and fixed hardware functions [3]. The memory controller brings together all the components and the caches together to provide data to every part of the system [3].

2.2.1 Programming in OpenCL

OpenCL is an industrial standard framework developed by the Khronos Group and used for writing programs that can be executed in heterogeneous systems containing OpenCL-compliant hardware as the device. OpenCL was built on the notion, "Write once, run on anything" which is similar to Java's "Write once, run everywhere" [39]. In other words, OpenCL enables portability across multiple OpenCL-compliant devices. The relevant tools to compile and run OpenCL code is provided by the vendor of OpenCL-compliant devices. Examples of such tools are AMD APP SDK and Xilinx SDAccel Development Environment.

OpenCL provides two compilation options - the traditional offline compilation and the runtime compilation [43]. Through runtime compilation, OpenCL allows execution on new hardware not available to the developer of the OpenCL program without the need to recompile the main application [43]. Through runtime compilation, OpenCL provides an interface to enumerate the available devices connected to the host i.e. CPU. On enumerating and selecting the required set of devices, OpenCL encapsulates the devices' details into a container called context. Device management is done through the context object [43].

Identical to CUDA, the functions that are executed in the devices are called kernels.

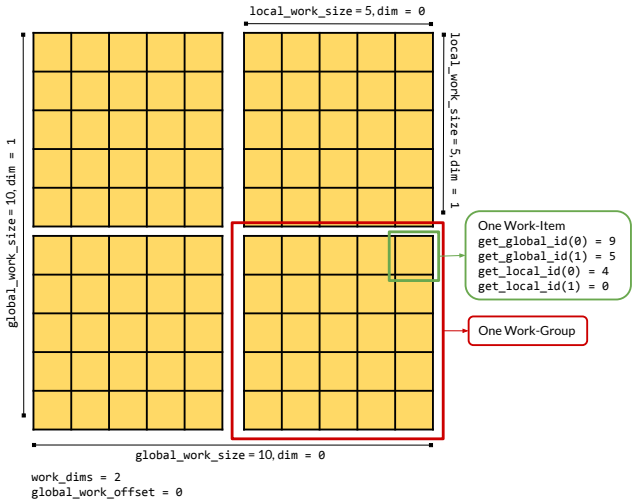


Figure 2.6: Example Organization of OpenCL Work-Items and Work-Groups

The given kernels are compiled runtime in a container called program [43]. After selecting the kernel in the program container, the appropriate arguments are associated to the kernels and dispatched to a queue data structure called the command queue [43]. The CPU offloads tasks to the device using the command queue. The device executes the functions in the command queue in the same order as they were queued.

The devices used for OpenCL contain at least one processor cores known as Compute Units which in turn contain one or more Processing Elements (PEs). The PEs are designed to execute SIMD instructions which thereby enables data-level parallelism [43]. From an AMD GPU perspective, OpenCL Compute Units are the same as the Compute Units (CUs) inside the GPU and each Processing Element (PE) is the SIMD engine inside the CU of the GPU. OpenCL also uses the concept of work-groups and work-items which are analogous to thread blocks and threads respectively in CUDA. The compute unit in an OpenCL device executes a work-group at a time [39] similar to a thread block executed by a SM in CUDA. Each work-item in a work group is assigned a global ID that uniquely identifies a work-item [39]. This can be obtained using `get_global_id(dim)` function call inside a kernel. The `dim` argument indicates the dimensionality of the data being processed [39]. The dimensionality is specified when queueing the kernel to the command queue. The minimum dimensionality is 1 and the maximum dimensionality is device dependent. Specifically, the following arguments are used to specify the work-item and work-groups while queueing the kernel:

- `work_dims`: The number of dimensions.

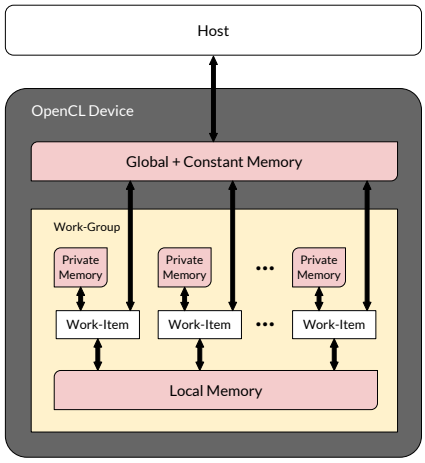


Figure 2.7: Overview of the OpenCL Memory Model

- `global_work_offset`: Offset to the global ID in each dimension.
- `global_work_size`: Total work-items in each dimension across all work-groups.
- `local_work_size`: Number of work-items in a work-group in each dimension.

Figure 2.6 shows an example organization of work-items and work-groups in a two-dimensional setting. Since a work-group can contain multiple work-items, `get_local_id(dim)` to get the local ID of a work-item to distinguishes it from the other work-items in the same work-group. After these arguments are specified, OpenCL determines the number of work-groups required before executing the kernel. If `local_work_size` is set to `NULL`, OpenCL will determine the best way to distribute the work-items based on the device architecture [39].

2.2.2 OpenCL Memory Model

OpenCL offers four types of memory - global memory, constant memory, local memory and private memory as shown in Figure 2.7. The global memory allows storage of data to be read or written by the entire device. This memory is designed to be large with high latency and may be shared with the host to transfer data to and from the host (CPU) [43]. This can be the Global Data Share (GDS) in a AMD GPU. Constant memory is similar to the global memory except that it is only read-only by the device. The location of the constant memory depends on the device architecture. Some devices may have a specific

memory location for Constant memory and the others may assign a part of the memory region in global memory to be constant memory [39]. The local memory is available to all the work-items in each work-group mainly for synchronization. This memory can be accessed by the work-items in a work-group faster than the global or constant memory but its size can be limited [39]. The LDS in each CU of a AMD GPU is the local memory. Lastly, private memory is a memory region for each work-item. Though it is limited in size, the private memory has the lowest latency access in the OpenCL memory model.

In order to use this memory model, the following address space qualifiers are available: `__global`, `__constant`, `__local` and `__private`. These address space qualifiers are used to define the address space of kernel arguments or other variables in the device code. `__global` is for global memory, `__constant` is for constant memory, `__local` is for local memory and `__private` is for private memory [39]. If a kernel argument or variable does not have an address space qualifier, then private memory is used for storage [39].

2.2.3 Example

Figure 2.8 shows an OpenCL with C++ bindings example to do vector addition in parallel. This file can be stored as a `.cpp` file and compiled using a C++ compiler such as `g++` with compiler options linking to the OpenCL library provided by the device vendor. To avoid redundancy, the function definition of `initvector()` function and the definition of `SIZE` and `START` have been ignored and are same as that of the CUDA example in section 2.1.3.

Before executing the actual computation, a preprocessing step is involved as OpenCL runtime compilation is used. First, the array of platforms available on the host is obtained, and the platform at index 0 is used in this example. Using this platform, a `Context` object is built which is in turn used to build the `Device` object. The device code that is hardcoded as a string is used to define a `Program` object. This `Program` object is compiled using the `Device` object to create a `Kernel` object. A `CommandQueue` object is declared using the `Context` and `Device` objects. After allocating the necessary GPU memory, the `CommandQueue` object is used to enqueue the transfer of input data using `enqueueWriteBuffer()` and the kernel as well. The kernel in this example is the `doSum` function and is launched with a 1-D data containing a global size of 1024 work-items. Each work-item runs the kernel by first obtaining its work-item ID using `get_global_id(0)` followed by the addition operation at the vector index equal to the work-item ID. While the addition is done in parallel, CPU is waiting on the response of the enqueue operation of `enqueueReadBuffer()` that copies the results from the GPU to the CPU. The results will be copied into the CPU memory once all the work-items complete the kernel execution.

```
1  #include "cl.hpp"
2  using namespace std;
3  using namespace cl;
4
5  const char *src =
6  "__kernel void doSum(__global int *a, __global int *b, __global int *
       sum) {" \
7  "    int i = get_global_id(0); " \
8  "    sum[i] = a[i] + b[i]; "\
9  "}";
10
11 int main(int argc, char *argv[]) {
12     cl_int err;
13     vector<int> h_a, h_b;
14     vector<int> h_sum(SIZE, 0);
15     Buffer d_a, d_b, d_sum;
16     initVector(h_a);
17     initVector(h_b);
18     vector<Platform> pl;
19     Platform::get(&pl);
20     cl_context_properties properties[] = { CL_CONTEXT_PLATFORM, (
           cl_context_properties)(pl[0])(), 0};
21     Context context(CL_DEVICE_TYPE_GPU, properties);
22     vector<Device> d = context.getInfo<CL_CONTEXT_DEVICES>();
23     Program::Sources source(1, make_pair(src, strlen(src)));
24     Program p = Program(context, source);
25     p.build(d);
26     Kernel k(p, "doSum", &err);
27     CommandQueue queue(context, d[0], 0, &err);
28     size_t bytes = sizeof(int)*SIZE;
29     d_a = Buffer(context, CL_MEM_READ_ONLY, bytes);
30     d_b = Buffer(context, CL_MEM_READ_ONLY, bytes);
31     d_sum = Buffer(context, CL_MEM_READ_WRITE, bytes);
32     queue.enqueueWriteBuffer(d_a, CL_TRUE, 0, bytes, h_a.data());
33     queue.enqueueWriteBuffer(d_b, CL_TRUE, 0, bytes, h_b.data());
34     k.setArg(0, d_a);
35     k.setArg(1, d_b);
36     k.setArg(2, d_c);
37     queue.enqueueNDRangeKernel(k, NullRange, 1024, NullRange);
38     queue.enqueueReadBuffer(d_sum, CL_TRUE, 0, bytes, h_sum.data());
39     return 0;
40 }
```

Figure 2.8: Vector Addition Example in OpenCL

Chapter 3

Enabling Rapid Construction of Arrival Curves from Execution Traces

3.1 Introduction

Verification of functional and non-functional properties in contemporary embedded systems are becoming more challenging and computationally intensive due to the inherent complexity and interconnectivity of such systems. Mathematical models and formal methods have been used in the past to derive performance metrics during the design phases [48, 37, 23]. Traditional formal methods have proved to effective on mapping worst-case behaviour for closed systems with trivial complexity to generalized mathematical representations. However, these approaches have failed to be useful and reliable for complex systems that have a high number of unpredictable interactions between the different components [7]. These limitations have turned the spotlight to processing execution traces in runtime analysis. The execution traces record the unpredictable behaviour in complex systems specifically the ones not considered during design time.

Arrival Curves offers a way to model the temporal behaviour of real-time systems. Coarsely grained approximations of arrival curves have been used by multiple frameworks to model worst-case behaviours and determine performance metrics by applying Network Calculus (NC) theory operations on these arrival curves [48, 37, 23]. Over time, various literature that focuses on the potential uses of fine-grained construction of arrival curves from execution traces in advanced techniques were published [20, 38, 24, 32]. However,

the caveat in the construction of such fine-grained arrival curves from large datasets of execution traces is that the process is computationally demanding which thereby results in the incomplete utilization of these curves in a real-world setting.

This chapter presents the algorithmic formulation for the construction of arbitrarily detailed arrival curves as a toolset by iterating over registered events in time-stamped execution traces. The algorithm also provides opportunities for data-level parallelism and experiments in this chapter show significant speedups when applying such parallelism to commodity parallel hardware such as GPUs.

The rest of the chapter is organized as follows: Section 3.2 overviews background of arrival curves and related work around the construction of arrival curve. Section 3.3 introduces some definitions to formalize the approach involved. Section 3.4 describes the core of the proposed algorithm for the construction of fine-grained arrival curves from execution traces including a discussion on the parallel approach involved. Section 3.5 shows the results of applying stress tests based on synthesized traces to the proposed algorithm with GPUs followed by an illustration of the constructed arrival curves using a QNX trace.

3.2 Background and Related Work

3.2.1 Analytical Arrival Curves

Arrival curves are represented as a function of interval time domain by providing the upper and lower limits on the number of registered events that occur in a system within a given time interval of length Δt [37]. Theoretically, arrival curves can be constructed from execution traces containing timestamps and events by sliding a time interval window of arbitrary length along the time axis while keeping track of the maximum and the minimum number of events occurring within each window. However, instead of using the execution traces, traditional approaches constructed the arrival curves by bounding any possible curve after analyzing generic behavioural patterns [7]. As a result, various scenarios were approximated using event pattern models that observe the period, jitter and delay of events (PJD models) [48, 37, 42]. Let the arrival curves constructed as a result be called *analytical* arrival curves.

3.2.2 Empirical Arrival Curves

However, the caveat behind using analytical arrival curves is that it will produce loose conservative approximations for unpredictable behavioural patterns that deviate from the general PJD pattern models [7]. This unpredictable behaviour arises due to the increasing complexity of embedded systems which has in turn increased the interest in alternative runtime analysis techniques that uses execution traces. Specifically, these techniques focus on producing fine-grained models that record the unpredictable workloads due to the unforeseen interactions between the various components in an embedded system.

Let the arrival curves constructed from execution traces of an embedded system be called *empirical* arrival curves. Though techniques to construct empirical arrival curves have potential uses in resource management [24, 32] and anomaly detection [38], it is a computationally intensive task as highlighted by the authors of [38] and therefore, remains overlooked in literature [7]. The only existing work that is designed to construct such curves from execution traces is the Real-Time Calculus (RTC) Toolbox [47]. However, this toolbox fails to process arbitrarily large execution traces [7]. This chapter will show the computational approach to construct the empirical arrival curves and also demonstrate the use of parallelism to overcome the computational intensive barrier.

3.3 Definitions

3.3.1 Traces Model

A trace is a chronological sequence of events occurring during the execution of a program. The trace may register multiple details about an event, such as an index, timestamp, and additional information pertaining to the functionality of the system. In this context, the index and the timestamp are the only parameters of interest.

Definition 1 *(**Event source**) An event source generates elements $ts_k, k \geq 0$, representing the timestamp of an event measured in an absolute time domain t. Timestamps are multiples of an atomic time unit ($ts_k \in \mathbb{N}$), and events are generated as time progresses ($ts_k < ts_{k+1}$). [7]*

Definition 2 *(**Trace**) A trace $TS = [ts_0, ts_1, \ldots, ts_{N-1}]$ is a finite sequence of N timestamps collected from an event source. [7]*

3.3.2 Empirical Arrival Curves

Definition 3 *(**Empirical arrival curve**) The pair of curves* $(\alpha^l(TS, \Delta t); \alpha^u(TS, \Delta t))$ *provide a lower and upper bound on the number of events seen in any time interval of length* Δt *in a trace* TS*. [7]*

To formulate an algorithm to construct empirical arrival curve, the term *quantized* arrival curves is used. Quantized arrival curves are approximations of empirical arrival curves calculated based on discrete buckets (i.e. windows of some time interval) of width r atomic time units.

Definition 4 *(**Quantized arrival curves**) Given a set of discrete buckets* Δt_i^B *of width* r *atomic time units* $(r > 1)$*, with bucket* Δt_i^B *enclosing all intervals* Δt *in the range* $ir \leq \Delta t < (i+1)r$*, with* $i \in \mathbb{N}^0$*. Quantized lower* $\hat{\alpha}^l(TS, \Delta t, r)$ *and upper* $\hat{\alpha}^u(TS, \Delta t, r)$ *arrival curves will provide a unique representative value for all intervals* Δt *enclosed in a bucket [7]. Quantized curves must comply with the following property:*

$$\forall r > 1 : \hat{\alpha}^l(TS, \Delta t, r) \leq \alpha^l(TS, \Delta t) \wedge \hat{\alpha}^u(TS, \Delta t, r) \geq \alpha^u(TS, \Delta t) \tag{3.1}$$

Equation 3.1 states that upper arrival curves and lower arrival curve need to be approximated from above and from below respectively [23]. The definition of quantized arrival curves also indicates a way to express a sequence of events occurring in a range of time intervals as a single bucket.

It is important to note that this definition considers the equivalence relations $\hat{\alpha}^l(\Delta t, r = 1) \equiv \alpha^l(\Delta t)$ and $\hat{\alpha}^u(\Delta t, r = 1) \equiv \alpha^u(\Delta t)$ as there is no quantization involved for $r = 1$. The remaining sections will be using the notations $\hat{\alpha}^l(\Delta t_i^B, r)$ and $\hat{\alpha}^u(\Delta t_i^B, r)$ to represent the quantized value of the lower curve and upper curve respectively for all intervals Δt in the bucket i.

3.4 Approach

3.4.1 Intuition

To construct the arrival curves, it is only required to check for intervals aligned to the events in the execution trace instead of checking all possible time intervals in the trace [32, 37].

This is general intuition behind the algorithmic formulation. Using this intuition, an iterative approach can be taken to construct the curves by iterating over the timestamps in the trace accumulating results obtained at each timestamp (pivot).

Figure 3.1 shows the processing flow of computing the arrival curves for a sample trace $TS = [3, 5, 6, 12, 16, 18]$ where each element denotes the timestamp of an event. This sample trace is represented illustratively on the top of the figure using absolute time axis t horizontally with red arrows representing the timestamps. This processing flow involves mapping each timestamp ts_k in the absolute time domain t to the interval time domain Δt using a reference timestamp ts_p, with $p \in [0, N-1]$. This mapping is in accordance with a function $M : t \rightarrow \Delta t$:

$$M(ts_k, ts_p) = ts_k - ts_p, \text{with } p \leq k \leq N-1 \tag{3.2}$$

During each iteration, a new distribution of events, p, is generated in the Δt domain and the local lower curve $\hat{\alpha}_p^l(\Delta t_i^B, r)$ and local upper curve $\hat{\alpha}_p^p(\Delta t_i^B, r)$ is computed for each pivot. These local curves are obtained by computing the minimum (lower bound) and maximum (upper bound) on the number of events occurring in time intervals measured from the pivot, which is the origin in the interval time domain Δt, to the limits of each interval bucket Δt_i^B. This process of computing local curves is repeated for the next succeeding timestamp which will be the new pivot.

In the case of $r = 1$ (no quantization), the following rules are used to compute the local curves [37]:

- The local lower value $\alpha_p^l(\Delta t)$ is the number of events in the interval $(0, \Delta t]$ in the current iteration.
- The local upper value $\alpha_p^u(\Delta t)$ is the number of events in the interval $[0, \Delta t)$ in the current iteration.

According to the definition of quantized arrival curves, a unique value must be assigned to every time interval lengths within a bucket for $r > 1$ to ensure that quantized lower and upper curves computed, as a result, approximates the curves computed using $r = 1$ from below and above respectively. This condition is fulfilled using the following rules [7]:

- The local lower value $\hat{\alpha}_p^l(\Delta t_i^B, r > 1)$ is the number of events in the interval $(0, ir)$ in the current iteration.

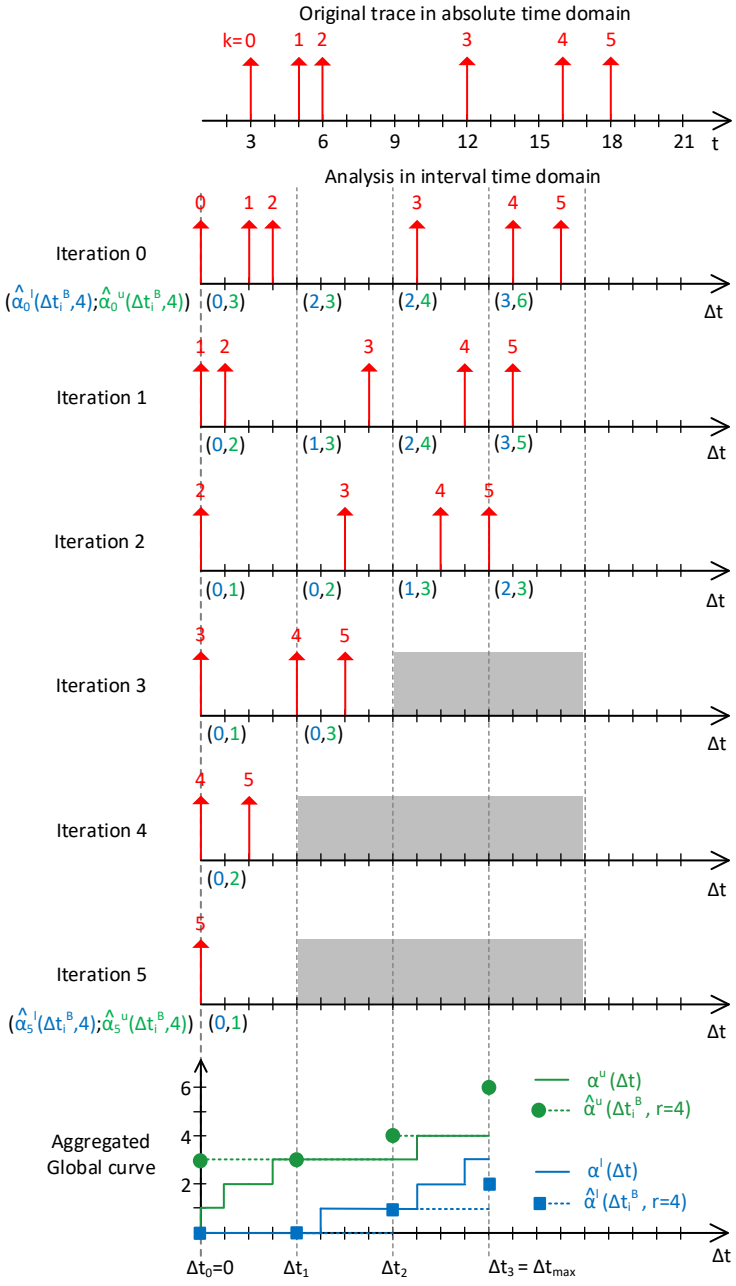


Figure 3.1: Processing Flow Example for Computing the Arrival Curves [7]

- The local upper value $\hat{\alpha}_p^u(\Delta t_i^B, r > 1)$ is the number of events in the interval $[0, (i + 1)r)$ in the current iteration.

The above rules will be used to compute the arrival curves. The example in Figure 3.1 has r set to 4 time units and another parameter, Δt_{max} set to $3r$. Δt_{max} is the maximum timespan of interest [7] and is, by default, less than or equal to $ts_{N-1} - ts_0$. The grey dashed vertical lines in the interval time domain shown in the figure separate each interval within a single bucket. The tuple shown below the Δt axis and right of the dashed vertical lines contains the lower and upper bound on the number of events for each bucket. For example, during iteration 0, the pivot is at timestamp $t = 3$ and hence, in the interval time domain, it is at the origin. Using the rules above for $r > 1$, the $\hat{\alpha}_0^l(\Delta t_0^B, 4)$ and $\hat{\alpha}_0^u(\Delta t_0^B, 4)$ values in the first time interval Δt_0 $(i = 0)$ are 0 and 3 as shown in the appropriate tuple in the figure. Similarly, for the second time interval Δt_1 $(i = 1)$, $(\hat{\alpha}_0^l(\Delta t_0^B, 4)$, $\hat{\alpha}_0^u(\Delta t_0^B, 4))$ is (2, 3) and so on until $\Delta t_i = \Delta t_{max}$. These are the local values during iteration 0. This computation is repeated for each timestamp as the pivot and the global lower and upper curves are obtained by taking the maximum and the minimum number of events over all the local curves. This is illustrated using the last plot at the bottom of Figure 3.1. The grayed areas in the figure denote that these areas are ignored when computing the global curves as no event occurs in those buckets. Also, it is important to know that the value of the curves at $\Delta t = 0$ is zero [37].

3.4.2 Algorithm

Algorithm 1 shows the pseudo-code of the formulated algorithm to construct quantized arrival curves with given arguments TS, $r > 1$, Δt_{max} and W where W is the length of the array `IntervalToBucket` which stores the number of events that occur within the maximum timespan Δt_{max}. If the minimum delay d_{min} between consecutive events in TS is known, then $W \geq (\Delta t_{max}/d_{min})$. The results of local curves during each iteration are stored in `LowLoc` and `UppLoc` and the aggregated results are stored in `LowGlob` and `UppGlob`. `INT_MAX` in the pseudo-code is any arbitrarly large positive integer.

The outermost loop in the algorithm is used iterate over the timestamps in TS and perform the computations for each timestamp. Initially the local lower and upper arrays are initialized to `INT_MAX` and -1 respectively during each iteration of this loop. Loop 1 deals with populating `IntervalToBucket` by mapping each timestamp $ts_k \geq ts_p$ to a bucket computed using its distance from the pivot timestamp ts_p. This means that at the end of Loop 1, `IntervalToBucket` will be populated with value i for all timestamps

Algorithm 1: Construct Quantized Arrival Curves from a Trace [7]

Input: TS, $\Delta t_{\max}$, r, W

```
1  nBuckets ← ⌊Δt_max / r⌋
2  LowGlob[1:nBuckets] ← INT_MAX
3  UppGlob[1:nBuckets] ← 0
4  LowLoc[nBuckets], UppLoc[nBuckets], IntervalToBucket[W]
5  for p ← 0 to N − 1 do
6      h ← 0, LowLoc[1 : nBuckets] ← −1, UppLoc[0 : nBuckets] ← −1
7      // Loop 1: Compute quantized values of intervals
8      for k ← p to N − 1 do
9          IntervalToBucket[h] ← ⌊(TS[k] − TS[p])/r⌋
10         h = h + 1
11         if (h ≥ W or IntervalToBucket[h] > nBuckets) then
12             break
13         end
14     end
15     // Loop 2: Find local upper and lower value for each bucket Δt_i^B
16     for i ← 1 to nBuckets do
17         if i ∈ IntervalToBucket then
18             UppLoc[i] = (index of last element equal to i)
19             LowLoc[i] = (index of first element equal to i) - 1
20         end
21     end
22     // Loop 3: Fill gaps in local upper curve and similarly for local lower curve
23     for i ← 1 to nBuckets do
24         if UppLoc[i] < 0 then
25             UppLoc[i] = UppLoc[i − 1]
26         end
27     end
28     // Loop 4: Update global curves
29     for i ← 1 to nBuckets do
30         UppGlob[i] = max(UppLoc[i], UppGlob[i])
31         LowGlob[i] = min(LowLoc[i], LowGlob[i])
32     end
33 end
```

considered such that $ir \leq \Delta t < (i+1)r$ in the interval time domain. Loop 2 extracts lower and upper values for the bucket Δt_i^B by looking at the the value i in `IntervalToBucket` and then updating `UppLoc` and `LowLoc` array appropriately at index i [7]. The last index of `IntervalToBucket` with bucket i is the local upper value and the first index of `IntervalToBucket` with bucket i minus 1 is the local lower value [7]. The subtraction of 1 when getting the local lower value is done to adhere to the rules of the intervals stated in the previous section.

However, values in `IntervalToBucket` may not contain all the buckets in the range from 1 to $nBuckets$ if the timestamps in TS are separated by more than r time units thereby, not updating all the indices of `UppLoc` and `LowLoc`. This is taken care of by Loop 3 which extrapolates the values to fill the gaps where the indices have not been updated [7]. Loop 3 only shows the operation on `UppLoc` where the missing values are filled with the previous maximum value. Similarly, another loop can be added to fill the gaps for `LowLoc` by iterating backwards from $nBuckets$ to 1 and updating the gaps with the next minimum value [7]. Lastly, Loop 4 takes care of updating the global curves index-wise by comparing the current global value with the local value and updating appropriately.

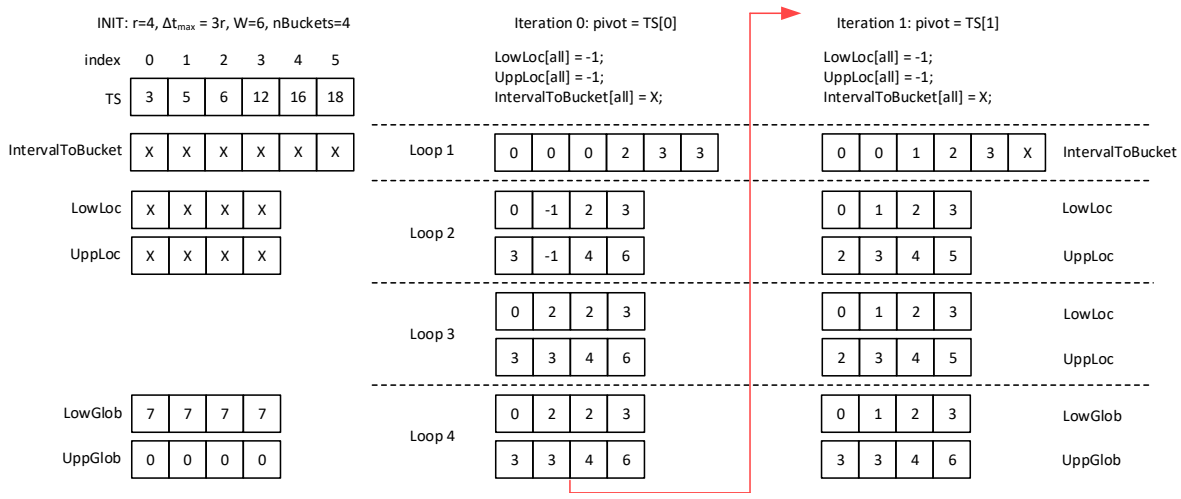


Figure 3.2: Illustration of the Algorithm Execution for the First Two Iterations over the Example Trace [7]

Figure 3.2 shows the execution of this algorithm for the first two iterations in the example trace $TS = [3, 5, 6, 12, 16, 18]$. The array values shown in the figure are the results after completing the execution of each loop. As discussed above, Loop 1 populates `IntervalToBucket` which is then used by Loop 2 to populate `UppLoc` and `LowLoc`. As mentioned earlier, not all the indices will be updated by Loop 2, and hence, Loop 3 fills the gaps. Lastly, Loop 4 updates the `UppGlob` and `LowGlob`. To ensure that the algorithm

follows the intuition discussed under Section 3.4.1, the values in `UppGlob` and `LowGlob` for iterations 0 and 1 in this figure are the same as that of the tuples in iterations 0 and 1 of the processing flow example in Figure 3.1.

3.4.3 Parallel Approach

This algorithm was formulated with the idea to transform the computations into embarrassingly parallel workloads. Such workloads are usually independent and need little or no communication between them.

Redesign: Loop 3 can be modified to perform the filling of gaps on the global curves instead of the local curves. However, this adds extra computation to Loop 2 as Loop 4 will be merged with Loop 2 to update global curves and also, add "markers" in the global curves to indicate the beginning and the end of gaps. These "markers" can then be used by Loop 3 to fill the gaps appropriately. This redesign allows Loop 3 to be moved outside the outermost loop thereby, adding potentially more parallel code, and the CPU can wait on the GPU to complete all parallel computations before copying the results from the GPU and then execute Loop 3.

Loop 1: From the pseudo-code in Algorithm 1 and the illustration of the arrival curve construction shown in Figure 3.2, each iteration of Loop 1 operate independently of one another for a given timestamp pivot. Using loop-based parallel pattern [11], Loop 1 can be transformed into a CUDA or OpenCL kernel.

Outermost Loop: As the update to the global curves for an event in a trace depends on the results of processing the previous events, the outermost loop in this algorithmic formulation cannot be parallelized. Hence, for each iteration of the outermost loop, kernels can be launched or enqueued into the command queue.

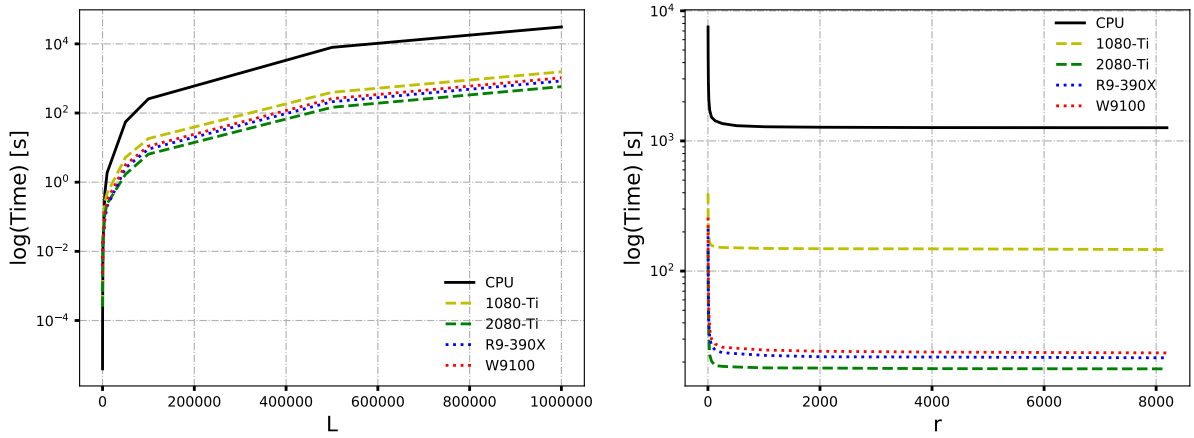
Loop 2: Loop 2 involves data dependencies between iterations. The operation can be described as a form of stream compaction i.e. given a vector, utilize a subset of this vector and output into another vector. This can also be parallelized in a kernel but not all threads or work-items will be active or executing the same instructions as only a subset of the input vector is used. Hence, the acceleration obtained from Loop 2 kernel will depend on the size of this subset which in turn depends on the distribution of the timestamps in TS and r. However, this drawback can be nullified as long as the overall parallelism achieved in the algorithm for a given set of parameters is large enough to overcome the overall sequential iterations .

Parallelism Factor: The loops that can be parallelized as discussed above involve parallelism over L and the number of buckets which depends on r. The number of buckets

increases as r decreases. Hence, the key factors to obtain acceleration via parallelism are L and r. Intuitively, a higher value of L, a lower value of r or a combination of both results in a higher number of iterations to operate on, thereby providing more opportunities for parallel computing.

3.5 Experiments

This section presents the execution time evaluation of sequential and parallel versions of the algorithm with GPUs on traces synthesized using a set of assigned values for L followed by the illustration of the arrival construction results on a QNX trace.



(a) Arrival Curve Construction Setup 1

(b) Arrival Curve Construction Setup 2

Figure 3.3: Arrival Curve Construction Execution Time Evaluation on a Synthesized Trace

Using Synthesized Traces on GPUs

The sequential version of the algorithm was implemented in C++ and executed on an 8-core Intel i7-3820 CPU 32GB of RAM. To demonstrate that the speedup obtained also depends on device specifications, two GPUs are used for each programming model. The CUDA implementation was executed on a Nvidia GeForce GTX 1080 Ti GPU and the recently released, Nvidia GeForce RTX 2080 Ti GPU which has relatively higher number of CUDA cores and better memory bandwidth. The OpenCL implementation was executed on an AMD R9-390X GPU and an AMD FirePro W9100 GPU. Though both GPUs have the

same number of stream processors, the AMD R9-390X GPU has better memory bandwidth. The speedup is calculated as the ratio between the sequential CPU execution time to the GPU execution time. The setups used in this case are the following:

Table 3.1: Arrival Curve Construction Speedup Results

Constant Parameters	Variable Parameter	Variable Values	1080 Ti Speedup	2080 Ti Speedup	R9-390X Speedup	W9100 Speedup
$r = 2$	L	10	0.015	0.016	0.002	0.002
		100	0.052	0.061	0.032	0.032
		1000	0.389	0.669	0.810	0.747
		5000	1.793	3.956	5.798	3.564
		10000	3.913	7.874	9.387	6.193
		50000	10.612	32.769	21.325	17.455
		100000	14.076	39.708	28.308	23.269
		500000	19.921	54.087	37.346	30.380
		1000000	19.798	53.174	36.423	29.528
$L = 500000$	r	2	19.000	51.594	35.564	29.217
		4	15.914	53.225	38.133	32.268
		8	13.294	56.761	43.051	36.045
		16	11.546	67.128	51.552	42.058
		32	10.440	75.649	55.981	48.238
		64	9.650	74.924	57.015	50.638
		128	9.312	76.090	57.855	51.921
		256	8.967	73.457	57.662	52.367
		512	8.679	71.525	56.458	51.300
		1024	8.614	71.267	57.144	52.003
		2048	8.583	70.904	58.090	52.725
		4096	8.550	71.224	58.001	53.133
		8192	8.636	71.313	58.672	53.805

- **Setup 1** has varying L with $r = 2$ and the results are shown in Figure 3.3a.
- **Setup 2** has varying r with $L = 500000$ and the results are shown in Figure 3.3b.

Each setup is iterated 10 times to obtain error bounds on the timing measurements. However, the error bounds of each curve in the plots are small and do not overlap with the other curves. Hence, in order to make the plots presentable, the error bounds are not included. Instead, the worst-case (maximum) timing values are used to plot the curves. The plots show that using the parallel versions provide a significant improvement in the execution time of the arrival curve construction for both setups with GeForce RTX 2080 Ti giving the best relative performance. Among the OpenCL devices, the R9-390X provides

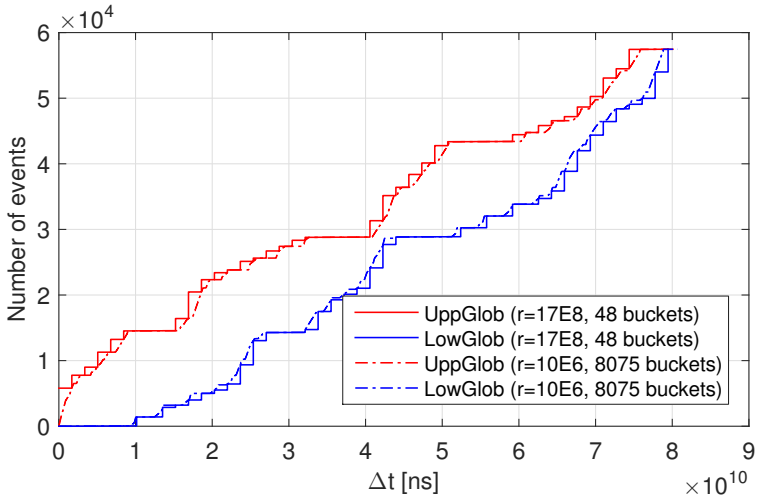


Figure 3.4: Arrival Curves Constructed from a Sub-Trace with Different Bucket Widths [7]

the best performance. Table 3.1 tabulates the speedups offered by each GPU. For $L < 5000$, GPUs do not offer speedup as the data transfer between the GPU and the CPU consume more time than the actual computation. Hence, the sequential execution can be preferred for these cases. However, as L increases, the computation becomes more intensive thereby making the data transfer less significant when compared to the overall execution thereby, offering more speedups. On the other hand, the number of buckets to operate on decreases as r increases resulting in a more coarser construction of arrival curves. This results in lower time consumption for higher values of r as shown in Figure 3.3b. However, acceleration can be obtained even for large values of r as long as the value of L is large enough to provide parallelism opportunities as shown in the table.

Using a QNX Trace

The QNX Real-Time Operating System (RTOS) is used in many safety-critical systems and is equipped with an advanced logging facility, *tracelogger* that enables detailed tracing of the various QNX activities on any system along with timestamps and identifiers for the type of event [7]. For this evaluation, we used a trace containing 19 types of events from an ARM Cortex-A8 processor running QNX RTOS and executing a *sequential sense-process-send* (SSPS) application, which mimics a data collection system sending data from sensors to a central hub [7]. This trace was then split into 19 sub-traces, one per each event type. Ideally, each sub-trace should map to arrival curves that are in accordance with the model of recurrent behaviour for the corresponding event type [7].

The arrival curves constructed from one such sub-trace using two different bucket widths r (measured in time units of the timestamps) is illustrated in Figure 3.4 and the curves are plotted in the same way as that of the arrival curves' illustrations in existing literature [37]. From the figure, we can see that the curves obtained using the larger bucket width ($r = 17 \times 10^8$) bounds the curves with lower bucket width ($r = 10 \times 10^6$) correctly. Though small values of r enable construction of fine-grained arrival models from the trace, it is not always preferable as it can result in over-fitting of the data in some cases. For example, if the bucket width is smaller than the minimum distance in time units between events then, the global curve may contain unnecessary buckets [7]. Hence, the bucket width r can be a tunable parameter to find the best trade-off among resource utilization and performance [7].

Chapter 4

Acceleration of Mining Arbitrary Regular Specifications from Execution Traces

4.1 Introduction

Characterizing and evaluating the temporal behaviour of complex programs has become an important task in many modern computer systems. Temporal properties specify the chronological occurrence of events while the program executes. However, in many practical applications, the programs are subject to continuous updates and are becoming increasingly complex, which leads to poor documentation and a general lack of formal specifications for their temporal properties. In this context, methods for dynamic specification mining, i.e., inferring temporal properties from execution traces, are gaining significant attention from multiple application domains [25]. Dynamically mined specifications are useful for software testing [13], automatic verification [14], anomaly detection [10], debugging [16], among other applications. In particular, in the embedded systems domain, programs that violate a predefined temporal behaviour during execution (out-of-order execution or delayed responses) can lead to major system failures. Moreover, if the program is part of a safety-critical application, then the timing violation can have catastrophic consequences for the environment and the user.

Dynamic temporal property miners identify a set of specifications that are satisfied by traces with respect to certain criteria. In general, miners receive an abstract description of critical and commonly occurring behavioural patterns, and then they look for specifications

of that form occurring in an input trace. Most literature on mining temporal specifications focuses on the qualitative notion of time; that is, the provided specifications only describe a chronological order of events without considering the actual time that passes between events. Recent work presents a framework tailored for real-time systems capable of mining properties with a quantitative notion of time, i.e., explicitly considering the length of the time interval between events. Independent of the particular scope, many approaches for mining temporal properties infer properties described in regular expression using some regular language class of automaton [30, 29].

This chapter presents a general framework that uses regular languages to mine specifications dynamically from execution traces and program logs. The core of the algorithm consists of mapping a generic regular expression template to multiple state machines that can be executed in parallel to mine concrete patterns while traversing the events registered in a trace. As presented, the core is general enough to be used as a miner for arbitrary patterns based in regular expressions, such as Timed Regular Expression (TRE) [30] and Nested Word (NW) [29]. We evaluate the utility of our approach using two case studies - mining TREs and NWs on synthesized traces to show that higher speed-ups can be achieved when mining arbitrary TRE and NW patterns from large traces. The algorithm is platform-independent and can be used to support other regular language class of automaton.

The rest of the chapter is organized as follows: Section 4.2 overviews related work around the development of computational frameworks for mining regular specifications. Section 4.3 introduces definitions and background to formalize our approach. Section 4.4 describes the core of the proposed algorithm for dynamic mining of specifications, including illustrations of the processing flow when mining TREs and NWs. Section 4.5 shows the results of applying our framework to TRE and NW mining case-studies, including stress tests based on synthesized traces using GPUs and illustration of the mining results using real-world datasets.

4.2 Related Works

The approach proposed in this paper involves dealing with a variety of challenging problems. Two of the major problems include scalable specification mining and using parallel pattern mining on datasets. Works that propose scalable approaches to mine specifications from execution traces are [26, 27, 15, 6, 41, 35, 22].

The work in [41] presents scalable specification mining using automata based abstractions limited to static analysis. [26] proposes a scalable technique to mine only tempo-

ral patterns for digital circuits using Linear Temporal Logic (LTL). The work under [15] presents scalable temporal specification miners evaluated on open-source Java programs. Similarly, specification mining with a focus on object-oriented programs is presented in [35]. A novel API specification mining architecture proposed by Lo et al. [27] presents an approach to learn probabilistic Finite State Automata after filtering erroneous traces to form specifications. A scalable Finite State Machine (FSM) inference technique to mine specifications using traces from the interaction of software libraries with the real software is proposed in [22]. A similar FSM inference technique discussed in [28] builds a guarded finite state machine from execution traces of object-oriented programs.

To summarize, unlike the works mentioned above, this paper presents an accelerated generalized framework to mine specifications in the form of regular expressions from execution traces relevant to the regular language used. Works about performing pattern matching using multiple regular expression in a parallelized manner are discussed in [8, 51, 33, 36]. The core of these works focuses on building one Deterministic Finite State Machine (DFSM) efficiently for a large number of regular expressions and use it for string matching in parallel. The main application for these papers is Network Intrusion Detection System. Similarly, pattern matching using TRE for runtime monitoring has been discussed in [46]. However, these works present *pattern matching* over a given dataset which is different from *pattern mining*. Pattern mining is the process of applying techniques on large amounts of data to extract patterns [18], and pattern matching is the process of comparing a detected pattern with a predetermined set of stored patterns [49]. This implies that pattern mining uses pattern matching to extract patterns [44]. Hence, the works presented in the papers mentioned above talk about efficiently detecting patterns in a data stream using regular expression matching engines with predetermined concrete instances. On the other hand, this chapter uses regular expression templates that contain variables to mine or extract patterns in parallel from large datasets.

4.3 Background and Definitions

4.3.1 Execution Traces Model

Definition 5 (Event) *An event corresponds to the occurrence of a certain functionality during the execution of a program. Each event has an event value that identifies its type.*

Definition 6 (Trace) *A trace* $T = [e_1, e_2, \ldots, e_{N-1}]$ *is a chronological sequence of events* e_i *occurring during the execution of a program. The trace may register multiple details*

about an event, such as the event type, timestamp, and additional information pertaining to the functionality of the system.

Definition 7 (Alphabet) *Assuming that a trace T will contain information about a finite set of event types, the alphabet Σ is the finite set of unique event types registered in the trace.*

Definition 8 (Alphabet size) *For a given trace T with alphabet Σ, the alphabet size $|\Sigma|$ is the number of event types registered in the trace.*

Consider the example trace $T_S = \{A,\ B,\ B,\ C,\ B,\ A\}$, registering a sequence of events that occurred in chronological order during the execution of a program. The first registered event corresponds to an event of type A, the second to an event of type B, and so on. The events as represented here just indicate the order and type of event but there can be traces with additional details about registered events according to the application context (e.g., timestamp, source, functionality, etc.).

For the example trace T_S, the alphabet is $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\} = \{A, B, C\}$, and the alphabet size is $|\Sigma| = 3$. In general, we will assume that the alphabet remains the same for multiple traces if the source of those traces (i.e., the program executed) is the same.

4.3.2 Regular Language

The regular language class of automaton provides a technique to specify patterns in the form of regular expressions [19]. Regular expressions provide a way to express the patterns for any system specification, and they can be converted to a DFSM. We can thus use DFSM representations to specify interesting patterns to be mined from an execution trace [30].

The proposed mining framework takes a template of a regular expression to generate a DFSM that can be used to mine temporal properties of interest. The following definitions introduce necessary terminology that we will use to formalize the proposed mining algorithm.

Definition 9 (Event variables) *An event variable is an atomic proposition in a regular expression that can take any event value from the trace alphabet Σ.*

Definition 10 (Template) *A template Π is a regular expression where all atomic propositions are either event variables and/or time intervals.*

A template is a generic representation of the patterns to be mined in a trace. Event variables in a template are essentially placeholders for specific event types that appear in a trace. For example, considering the event variables α and β, the template $< \alpha; \beta >$ represents the target pattern of α followed by β. Here, α and β can take any value from the alphabet Σ. Templates can also specify regular expressions with arbitrary complexity, including patterns that occur within defined time intervals. Two particularly interesting forms of regular expressions that have gathered interest recently are TRE [30, 46] and NW [29, 1]:

- TRE example: $< \alpha; \beta > [x, y]$, where x and y specify a time interval of length $y - x$ time units. The template represents the pattern α followed by β such that $t_\beta - t_\alpha \leq y - x$, where t_α and t_β represents the time of occurrence of events α and β with respect to a reference clock.
- NW example: $(\langle \alpha^n . \rangle \beta^n)$ where α represents recursive call events occurring n times followed by n recursive return events β.

Definition 11 (Variable Set) *The variable set P is the set of event variables in the given regular expression template.*

Definition 12 (Dimension) *The dimension $d = |P|$ is the number of event variables in the given regular expression template.*

Definition 13 (Instance) *For a given template Π, π_k is an* instance *of Π for a trace T if π_k is a pattern expressed by Π with the event variables substituted with the corresponding constants or literals from the alphabet of T. Subindex k specifies a unique identifier for the instance for a given permutation of the alphabet in the event variables.*

Definition 14 (Mapping) *A mapping is a function $f : P \to \Sigma$, where P is the finite set of event variables in the regular expression template and Σ is the alphabet of the trace.*

It is important to note that $|\Sigma| \geq d$. If $|\Sigma| < d$, then there are more variables in the template than $|\Sigma|$ and not all variables can be mapped to the events in Σ to form an instance. To illustrate the previous definitions, let us consider the earlier example trace T_S and the TRE template $\Pi =< \alpha; \beta > [x, y]$. In this case, the variable set is $P = \{p_1, p_2\} = \{\alpha, \beta\}$, and thus $d = |P| = 2$. The mapping function f maps each event variable in P to each event type in the alphabet $\Sigma = \{A,\ B,\ C\}$, resulting in the instances shown in Table 4.1. The use of templates and the mapping function highlights the difference between pattern mining and pattern matching.

Table 4.1: Mapping of Template $< \alpha; \beta > [x, y]$ to Alphabet $\Sigma = \{A, B, C\}$

Instance	α	β	Instance π
π_1	A	A	$< A; A >$
π_2	A	B	$< A; B >$
π_3	A	C	$< A; C >$
π_4	B	A	$< B; A >$
π_5	B	B	$< B; B >$
π_6	B	C	$< B; C >$
π_7	C	A	$< C; A >$
π_8	C	B	$< C; B >$
π_9	C	C	$< C; C >$

4.3.3 Dominant Properties

The set of instances generated by mapping the template with Σ contains all possible $|\Sigma|^p$ permutations of event types. These generated instances are used as patterns to mine concrete specifications from the trace. However, these instances can contain both frequently occurring patterns and some patterns that may be present only a few times. The intuition behind specification mining is that frequent patterns occurring in the traces are most likely to be true [26]. Hence, we use a ranking component that filters the set of mined patterns to consider only the dominant instances or specifications of a given template. The concepts of support and confidence are defined based on [25].

The ranking component used in this framework makes use of four parameters - Support ω_{π_k} - total number of times the pattern of an instance π_k was successfully mined in a trace, Support Potential Ω_{π_k} - total number of times the pattern of an instance π_k was *starting* to be mined in a trace, Confidence δ_{π_k} - ratio of ω_{π_k} to Ω_{π_k} and ϵ - an user-defined threshold on δ_{π_k} values. Ideally, the ranking component should be designed with these parameters such that it would evaluate and mine mainly the dominant patterns or properties that can help in determining specifications.

4.4 Approach

Figure 4.1 shows a high-level overview of the proposed framework for mining specifications using regular languages. The first stage involves the preprocessing of the input trace(s) and the template to generate a DFSM representation and then perform the mapping to

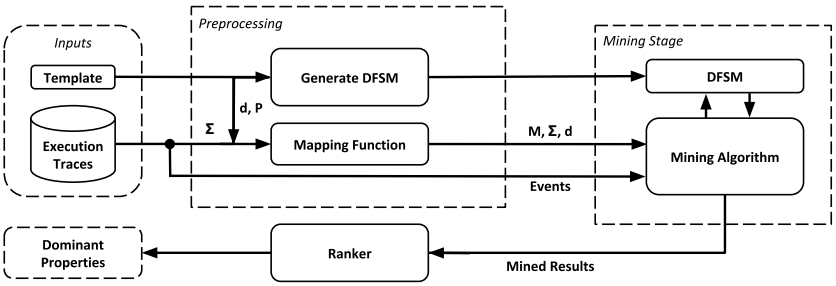


Figure 4.1: High-Level Overview of the Mining Framework

generate multiple instances associated with the alphabet values. During the mining stage, the mapping and the events from the input traces are passed to the mining algorithm which runs through the template instances using the generated DFSM to perform matching of specific patterns with concrete alphabet values. The mining generates counter values for each instance, which the ranker uses to filter dominant properties.

4.4.1 Inputs

The inputs to the framework are a trace or a set of traces and the template in the form of a regular expression. Apart from the event type that is necessary to obtain the alphabet of the trace, each registered event can be represented as a structure containing multiple fields of information about the event. For example, when mining TREs, each registered event must contain at least a field for its type and timestamp based on a reference system clock. The template can be given in the form of a string.

4.4.2 Preprocessing

The preprocessing stage consists of extracting information from the input traces and the template to generate the DFSM and the mapping function.

Figure 4.2 shows an example DFSM representation for the template $< \alpha; \beta > [x, y]$ introduced previously. A DFSM uses a transition function that takes as argument a current state and a transition symbol to return the next state [12]. In the case of mining TREs, the transition function for a DFSM should also keep track of the timing constraint in addition to the transition symbols. We use TRE for this example because it represents a

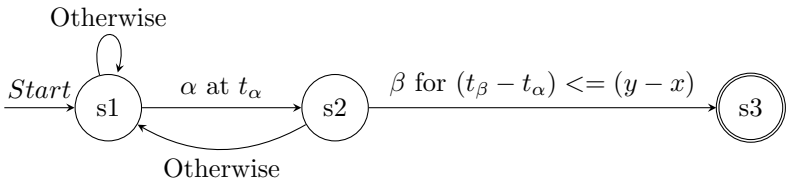


Figure 4.2: Illustrative DFSM for Template $< \alpha; \beta > [x, y]$

more general model that includes a quantitative notion of time [30]. A traditional regular expression can only provide a qualitative notion of time (i.e., the order of events) and could be directly obtained by removing the timing conditions from the transitions in a TRE. In the case of NWs, the transition functions should keep track of the number of calls and returns as defined for a Nested Word Automaton (NWA) [29].

Algorithm 2: A Transition Function for DFSM in Fig. 4.2

```
1  start_time ← NULL
2  Function transition(current_state, symbol, new_time):
3      if current_state = s1 then
4          if symbol = α then
5              start_time = new_time
6              return s2
7          end
8          else
9              return s1
10         end
11     end
12     if current_state = s2 then
13         if symbol = β & (new_time − start_time) <= (y − x) then
14             return s3
15         end
16         else
17             return s1
18         end
19     end
20 end
```

From a programming context, the transition function of a DFSM for a given template

can be represented using either conditional branches (if-then-else) or table-driven formats in the form of a transition table. Algorithm 2 shows an example transition function for the TRE template DFSM in Figure 4.2 in the form of conditional statements. The argument *symbol* represents the given transition symbol from the current state which is represented by the *current_state* argument. The argument *new_time* represents the timestamp for the occurrence of *symbol*. In the example DFSM shown in Figure 4.2, if the current state is s1 and the DFSM receives α at time t_α, then $symbol = \alpha$ and $new\_time = t_\alpha$. In this case, the transition function shown in Algorithm 2 would set *start_time* to t_α and return s2 as the new state. The *start_time* variable is used to check whether the timing constraint specified in the template is satisfied when transitioning from s2 to s3.

Algorithm 3: Pseudo-code for TRE **update_instance**()

```
1  Function update_instance(current_state, symbol, new_time, S_i, R_i):
2      current_state ← transition(current_state, symbol, new_time)
3      if is_accepting_state(current_state) then
4          // Success
5          S_i = S_i + 1
6          current_state ← get_start_state()
7      end
8      if current_state = get_start_state() then
9          // Failure
10         R_i = R_i + 1
11     end
12 end
```

There exist third-party compiler libraries that can generate such DFSMs. For example, the *Ragel State Machine Compiler* [31] can generate DFSMs with the minimum number of states. Also, Ragel provides the option to generate code in various formats such as binary search table-driven, flat table-driven, and branch-driven along with the feature to have customized transition functions. In our framework, the function `update_instance()` (shown in Algorithm 3) uses the transition function in Algorithm 2 to update the state and to increment the S_i and R_i counters that keep track of the number of successes and failures of an instance. The arguments *symbol*, *new_time*, *current_state* have the same representation as that of the transition function.

The mapping for the example trace T_S shown in Table 4.1 shows all possible instances. Since the example template explicitly utilizes two different variables - α and β, we are only interested in instances with $\alpha \neq \beta$. This means that instances π_1, π_5 and π_9, are not of

interest for the template and thus we can ignore them. In general, we can determine the number of useful instances for each event type as [30]:

$$N_\pi = d * \prod_{i=1}^{d-1}(|\Sigma| - i) \tag{4.1}$$

In the example used for Table 4.1, the set of instances that contain event type A are $\{\pi_2, \pi_4, \pi_6, \pi_8\}$, and the size of this set is equal to $N_\pi = 4$.

Table 4.2: 2D Mapping Array M for Instances of Interest

A	(π_2, α)	(π_3, α)	(π_4, β)	(π_7, β)
B	(π_2, β)	(π_4, α)	(π_6, α)	(π_8, β)
C	(π_3, β)	(π_6, β)	(π_7, α)	(π_8, α)

Table 4.2 shows a convenient reorganization of the information in Table 4.1. Each row in the new 2D mapping array M contains the information related to a particular event type from the alphabet Σ, listing all useful instances on which the corresponding event type appears. Each column in a row of M contains a pair (π_j, p_k), where π_j ($j < |\Sigma|^d$) corresponds to an useful instance where the event type for that row appears, and p_k corresponds to the k-th event variable in the template for which the event type appears for that instance. In general, M will have $|\Sigma|$ rows and N_π columns.

4.4.3 Mining Stage

The mining algorithm serves as the core part of the framework. The pseudo-code for the mining algorithm is shown in Algorithm 4. The algorithm receives the trace, the mapping array M, the alphabet Σ and the dimension d. `s` is a 1D array containing the current state of the DFSM for an instance. The size of this array equals the number of instances N_I, and each index of this array corresponds to a unique instance number. The success array `S` and the reset array `R` store the number of successes and the number of failures of a pattern identified by the unique instance number. `R` is called the reset array as a failure of an instance indicates "resetting" the current state of the instance to the start state to possibly mine same instances during the future processing of the trace. In the case of successful instances, the corresponding instance is set to start state after reaching the final accepting state. This is demonstrated in the example pseudo-code for `update_instance()` in Algorithm 3.

Algorithm 4: Pseudo-code for Mining Algorithm

Input: T, M, Σ, d

```
1  N_I ← |Σ|^d
2  s[1:N_I] ← get_start_state()
3  S[1:N_I] ← 0
4  R[1:N_I] ← 0
5  // Loop 1 over all the events in T
6  foreach e in T do
7  |  I_C[1 : N_I] ← FALSE
8  |  // Loop 2 over instances that considers event type e
9  |  foreach column in M[e] do
10 |  |  // Extract information from each element in M
11 |  |  v ← M[e][column].first  // Get instance
12 |  |  i ← M[e][column].second // Get template variable
13 |  |  update_instance(i, s[v], S[v], R[v])
14 |  |  I_C[v] ← TRUE
15 |  end
16 |  // Loop 3 over instances that does not consider event type e
17 |  for v ← 0 to N_I do
18 |  |  if I_C[v] = FALSE then
19 |  |  |  update_instance(NULL, s[v], S[v], R[v])
20 |  |  end
21 |  end
22 end
```

The mining process is implemented using nested loops. Loop 1 goes through all the events in the same order as provided by the trace T. Loop 2 goes through all instances associated with the type of event e. The reasoning behind is that, for an event type e, we need to substitute the type of e into each of the d event variables of the template. We achieve this by accessing the row in M corresponding to the type of event, represented as $M[e]$ in Algorithm 4. The data structures as shown in the pseudo-code are passed on to `update_instance()` which updates the corresponding instance and stores the new state in `s`. Based on the computation results, the corresponding entry in `S` or `R` is incremented. At the end of the mining process, `S` and `R` are passed to the ranker component to extract dominant properties.

Loop 3 is necessary to cover transitions when the current event type is not explicitly considered in an instance. For example, π_2 in Table 4.1 does not consider events of type C. Thus, when evaluating an event of type C, π_2 should reset to the initial state, and this is done by Loop 3. This processing flow is explained in detail using examples for a TRE and a NW below.

Processing Flow for TRE Mining

Let us consider the sample trace for mining TREs:

$$T_{TRE} = \{[0, A]; [2, B]; [3, C]; [5, B]; [7, C]; [8, A]\}$$

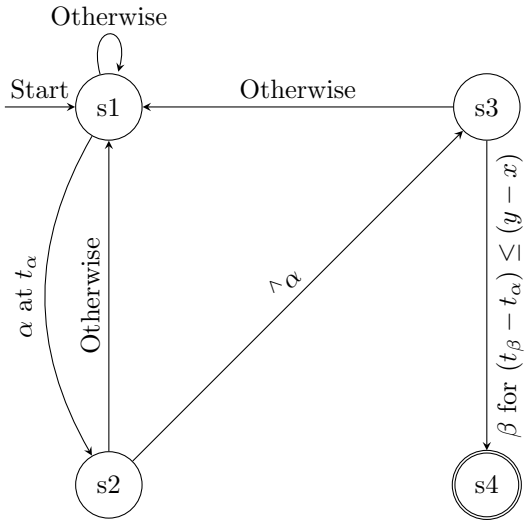
where each registered event has the form $[ts, type]$, with ts as timestamp in time units related to a global clock and $type$ is the type of event. For this trace, $|\Sigma| = 3$, $d = 2$, and the mapping array M is the same as in Table 4.2.

Let us also consider the TRE template:

$$< \alpha; (\hat{}\alpha); \beta > [0, 3]$$

representing the pattern α followed by something different than α and then followed by β, within an interval of three time units. The template uses the negation operator $\hat{}$ as stated in [30].

Figure 4.3a shows a representation of the template as a DFSM, and Figure 4.3b shows the execution flow of Algorithm 4 when mining using the given TRE template. Columns represent one iteration of Loop 1, and rows represent the execution of Loops 2 and 3 for each instance from the mapping table. Horizontal thick lines separate the processing flow and results obtained for each instance. For each iteration of Loop 1, the example execution



(a) DFSM for TRE $< \alpha; (\hat{}\alpha); \beta > [x, y]$

Time →

Instance		Loop 1	ts = 0, e = A	ts = 2, e = B	ts = 3, e = C	ts = 5, e = B	ts = 7, e = C	ts = 8, e = A
π_2	α = A, β = B	Loop 2	s2	s3		R		s2
		Loop 3			R		R	
π_3	α = A, β = C	Loop 2	s2		S		R	s2
		Loop 3		s3		R		
π_4	α = B, β = A	Loop 2	R	s2		R, s2		S
		Loop 3			s3		s3	
π_6	α = B, β = C	Loop 2		s2	s3	R, s2	s3	
		Loop 3	R					R
π_7	α = C, β = A	Loop 2	R		s2		R, s2	s3
		Loop 3		R		s3		
π_8	α = C, β = B	Loop 2		R	s2	s3	R, s2	
		Loop 3	R					s3

(b) Example Execution Flow for TRE Mining

Figure 4.3: DFSM Representation of a TRE Template and Execution Flow for TRE Mining

flow shows the state of each instance at the end of Loop 2 and Loop 3. The initial state for each instance is s1 as shown in Algorithm 4.

Let us describe the processing for instance π_2 while traversing the trace. Intuitively, instance π_2 performs pattern matching looking for an event of type A followed by any event of not type A and then, followed by an event of type B. Starting with the first event of type A and $ts = 0$, instance π_2 proceeds to s2 at the end of Loop 2, because event type A corresponds to α for that instance (see Table 4.1). In the next iteration of Loop 1, the event type is B, which makes instance π_2 to go to s3. The next event is of type C, which is not considered explicitly in instance π_2. This causes Loop 2 to end without any effect, and Loop 3 resets the instance taking it back to s1. The next event is of type B, which resets the instance to s1. In the processing flow for instance π_2, resets shown in blue are due to failed matching of event types considered in that instance, and resets shown in red represent patterns that are interrupted by the occurrence of event types that are not considered in the pattern of that instance. All types of reset increment the corresponding counter in `R` by 1. This processing and intuition applies to all instances.

In the execution flow for instance π_3, green fields indicate a path that leads to a success in the pattern matching for that instance. For the first event with $ts = 0$ and type A, instance π_3 proceeds to s2 at the end of Loop 2, because type A corresponds to α for that

instance. The next event is of type B, which is not considered in instance π_3. This makes loop 2 to end without any effect. Unlike instance π_2, Loop 3 will cause instance π_3 to go to s3 as the event type satisfies the transition from s2 to s3, i.e., ($\hat{}\alpha$). This highlights the use of the negation operation. In the next iteration of Loop 1, the event type is C and $ts = 3$ which makes instance π_3 to go to s4 as type C corresponds to β for this instance, and the pattern occurred within three time units. Since s4 is the accepting state, the corresponding counter in $\mathbf{S}$ is incremented by 1, and the instance goes back to s1.

In the processing flow of instance π_4 at $ts = 5$, the instance is reset and then proceeded to s2. This is because of event type B in that iteration being the first transition value for the DFSM, i.e., α of instance π_4. The reason for resetting an instance is to possibly mine future instances. In this particular case, the reset happens at an event that can start the mining of the instance. Hence, after reset, instance proceeds with $e = B$ and thereby, set to s2. Eventually, this instance succeeds at $ts = 8$.

Processing Flow for NW Mining

Let us consider the sample trace:

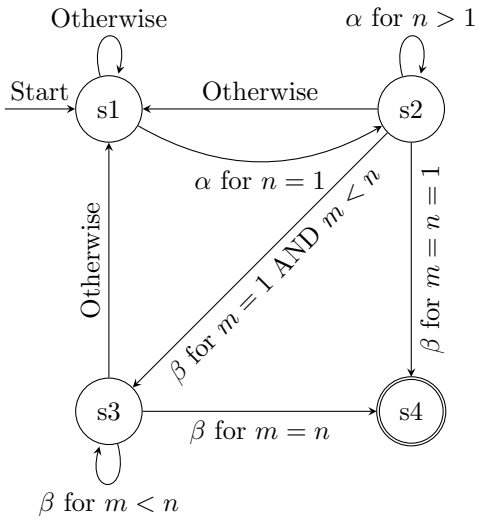
$$T_{NW} = \{A, B, A, C, A, B\}$$

and the template

$$[\langle\alpha]^n.[\rangle\beta]^n; n > 0$$

to illustrate the processing flow of NW mining. The template used here can be interpreted as: n calls of α followed by n returns of β. Since $\Sigma = 3$ and $d = 2$, the mapping array is the same as in Table 4.2. Similar to the processing flow for TRE, Figure 4.4a shows a representation of the template as a DFSM, and Figure 4.4b shows the execution flow of Algorithm 4 when mining using the given NW template.

As mentioned earlier, the goal of Loop 3 is to cover transitions when the current event type is not considered in an instance. This is applied to mining using NW to reset hanging Nested Word Automata (NWAs). Let us describe the processing for instance π_2 while traversing the trace. Intuitively, instance π_2 performs pattern matching looking for n occurrences of an event of type A followed by n occurrences of an event of type B. For the first event type A, π_2 proceeds to s2 as A is considered to be α in π_2. For the next event type B, π_2 proceeds to s4 and succeeds as B is considered to be the β in π_2. Similarly, π_2 proceeds to s2 for the next event of type A. However, since the next event of type C is not considered by π_2, Loop 2 does not have any effect on π_2. This causes Loop 3 to reset π_2 (shown in red). In other words, π_2 was "hanging" and Loop 3 resets it. Suppose



(a) DFSM for NW ($[\langle\alpha]^n.[\rangle\beta]^n$); $n > 0$

Instance		Time → / Loop 1	e = A	e = B	e = A	e = C	e = A	e = B
π_2	α = A, β = B	Loop 2	s2	S	s2		s2	S
		Loop 3				R		
π_3	α = A, β = C	Loop 2	s2		s2	S	s2	
		Loop 3		R				R
π_4	α = B, β = A	Loop 2	R	s2	S		R	s2
		Loop 3				R		
π_6	α = B, β = C	Loop 2		s2		R		s2
		Loop 3	R		R		R	
π_7	α = C, β = A	Loop 2	R		R	s2	S	
		Loop 3		R				R
π_8	α = C, β = B	Loop 2		R		s2		R
		Loop 3	R		R		R	

(b) Example Execution Flow for NW Mining

Figure 4.4: DFSM Representation of a NW Template and Execution Flow for NW Mining

Loop 3 does not exist, then π_2 would be "hanging" at s2 after type C has been processed and would take the third occurrence of event type A as another call for the nested word and increment n in its NWA. However, this is not a valid structure of the Nested Word as defined in [29]. The processing flow continues, and π_2 succeeds again after the reset. This processing flow applies to all instances.

4.4.4 Ranker

The ranker filters the dominant patterns or properties from the set of all mined patterns. The intuition behind this is that frequently occurring patterns are most likely to be true [26]. An index k in `S` and `R` are used to get ω_{π_k} and $(\Omega_{\pi_k} - \omega_{\pi_k})$ values respectively for an instance π_k. Using these values, we obtain the support potential Ω_{π_k} and then calculate the confidence δ_{π_k}. Using a user-defined filter threshold ϵ, the mined patterns or properties can be ranked and filtered to yield the dominant properties.

4.4.5 Parallel Approach

The mining algorithm was formulated with the notion to transform the computations into embarrassingly parallel workloads. Such workloads are usually independent and need little or no communication between them.

Loops 2 and 3 as Kernels: From the processing flow of TRE mining and NW mining shown in Figure 4.3b and Figure 4.4b respectively, we can see that instances operate independently of one another for a given event. This is also highlighted in the figures by the thick horizontal lines separating the instances. This means that each iteration of Loop 2 and Loop 3 in Algorithm 4 is independent of the other iterations, and each iteration perform the same function on different instances. This means that using loop-based parallel pattern [11], Loop 2 and Loop 3 can be transformed into two separate CUDA or OpenCL kernels.

Loop 1: As DFSMs are used and the current state of a DFSM when processing an event in a trace depends on the results of processing the previous events, Loop 1 in this algorithmic formulation cannot be parallelized. Hence, for each iteration of Loop 1, these two kernels can be launched or enqueued into the command queue.

Using DFSM Inside Kernels: These kernels will also involve using the transition function of a DFSM. However as branching affects the performance of work-items in a wavefront or threads in a wrap, it would be a good design decision to generate the DFSM code as a table-driven format thereby, keeping the instructions same. Minor experimental evaluations in OpenCL among the various DFSM formats offered by Ragel showed that the table-driven DFSM format with certain Ragel optimizations provided the best relative performance. Also, assuming the DFSM of a given template has small memory requirement, another good option would be to store the variables related to the DFSM in constant memory as these variables are written to the device memory by the host only once and read many times by the mining algorithm.

Parallelism Factor: Given that the iterations of Loop 2 and Loop 3 are parallelized, the acceleration obtained would depend on the number of instances processed independently. The number of instances as seen in Section 4.3 is $|\Sigma|^d$. Hence, the key factors to obtain acceleration via parallelism are $|\Sigma|$ and d. Intuitively, a higher value of $|\Sigma|$, d or both results in a higher number of instances to operate on, thereby providing more opportunities for parallel computing. This behaviour is demonstrated in the experiments section below.

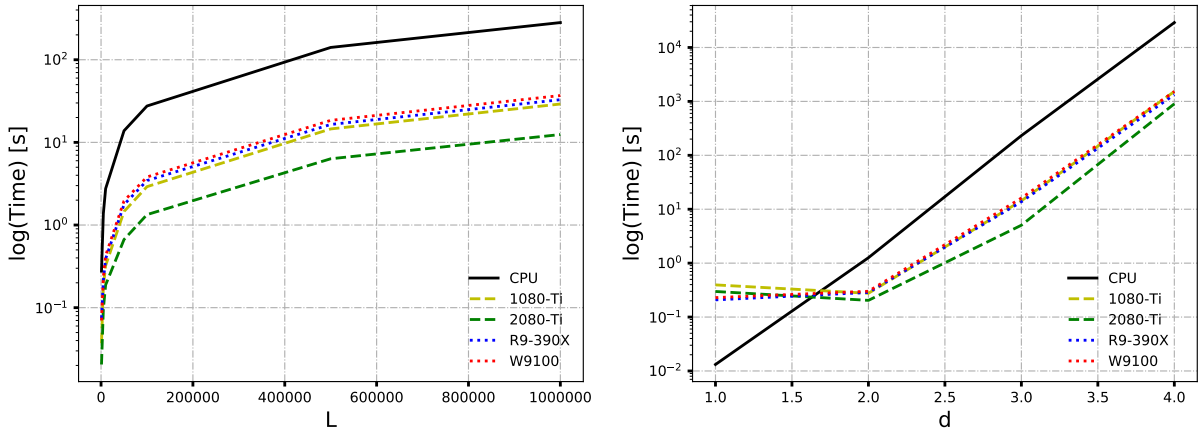
4.5 Experiments

Experiments were conducted on TRE mining and NW mining to observe the acceleration obtained via parallelism. For each mining technique, two experiments are presented. The first experiment presents the evaluation of mining execution time on synthetic traces for three different setups. The second experiment showcases the results of the mining on a real-world dataset. The calculation of speedup and the device setups used for the synthesized traces are same as that of the arrival curve construction experiments.

4.5.1 Case Study: Timed Regular Expression (TRE)

Using Synthesized Traces on GPUs

The setups used for this case study are the following:

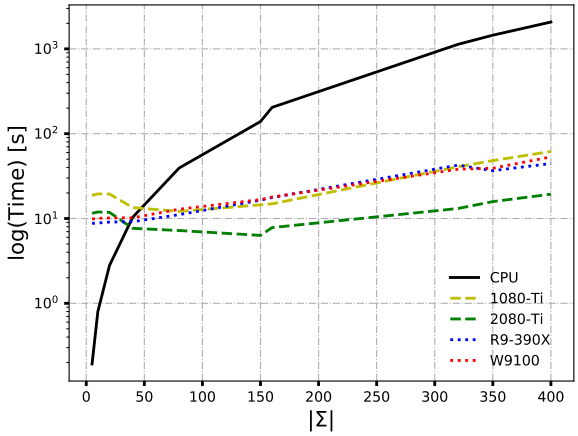


(a) TRE Mining Setup 1

(b) TRE Mining Setup 2

Figure 4.5: TRE Mining Execution Time Evaluation on Synthesized Traces (Setups 1 and 2)

- **Setup 1** has varying L with $|\Sigma| = 152$, $d = 2$ and the results are shown in Figure 4.5a.
- **Setup 2** has varying d with $L = 10000$, $|\Sigma| = 98$ and the results are shown in Figure 4.5b.
- **Setup 3** has varying $|\Sigma|$ with $L = 500000$, $d = 2$ and the results are shown in Figure 4.5c.



(c) TRE Mining Setup 3

Figure 4.5: TRE Mining Execution Time Evaluation on Synthesized Traces (Setup 3)

The TRE template

$$t_{TRE\_A} = (\hat{}(P|S)^*.(\langle P.\hat{}(P|S)^*.S.\hat{}(P|S)^*\rangle[0, 2000]))+$$

known as the *Alternating* template [30] is used for experiments where d (set to 2 in this case) is kept constant. This template is modified by adding or removing variables to evaluate for experiments where d is varied.

The plots show that using the parallel versions provide a significant improvement in the execution time of the mining for large values with Nvidia GeForce RTX 2080 Ti performing relatively and significantly better than the other devices. R9-390X provides the best performance among the OpenCL devices. $|\Sigma|$ and d are the key scaling factors in this parallel computation. More the alphabet size and/or the number of template variables more is the number of instances which thereby, enables significant improvement in execution time via parallelism as shown in Figures 4.5b and 4.5c. This allows mining on traces with large alphabet size using templates with several variables within a reasonable amount of time. On the other hand, the behaviour of L is linear over time, and the parallelized implementation provides an improvement in execution time for all values as seen in Figure 4.5a as the number of permutations is fixed and large enough for significant parallelism. This behaviour is also tabulated in Table 4.3 which shows the speedups obtained from each GPU. For setup 1, the speedups are almost consistent across all GPUs as L increases due to L being large enough with a large constant $|\Sigma|$. However, the speedups increase as the variable values in setups 2 and 3 increases. No speedup is obtained for low variable values as the data transfer dominates the time consumption. As shown in the table, the

Table 4.3: TRE Mining Speedup Results

Constant Parameters	Variable Parameter	Variable Values	1080 Ti Speedup	2080 Ti Speedup	R9-390X Speedup	W9100 Speedup
$d = 2$ $\|\Sigma\| = 152$	L	1000	7.453	13.489	3.704	3.954
		5000	7.670	14.323	6.450	5.859
		10000	8.531	14.622	7.063	6.463
		50000	9.487	20.709	7.820	7.061
		100000	9.460	20.610	7.984	7.221
		500000	9.683	22.239	8.547	7.627
		1000000	9.670	22.642	8.576	7.639
$L = 10000$ $\|\Sigma\| = 98$	d	1	0.033	0.044	0.063	0.057
		2	4.565	6.183	4.410	4.194
		3	16.158	45.875	16.866	14.368
		4	18.981	32.131	21.710	18.964
$d = 2$ $L = 500000$	$\|\Sigma\|$	5	0.010	0.017	0.022	0.020
		10	0.041	0.067	0.091	0.080
		20	0.143	0.236	0.307	0.275
		40	0.769	1.351	1.129	1.014
		80	3.249	5.462	3.547	3.062
		150	9.598	22.027	8.467	8.322
		160	13.710	26.142	11.556	11.416
		320	27.760	86.253	26.605	29.720
		350	30.109	91.650	39.572	37.068
		400	33.601	107.156	46.658	38.929

differences in speedups among GeForce GTX 1080 Ti, R9-390X and W9100 are small for a given variable value in these setups. On the other hand, GeForce RTX 2080 Ti stands out with significant jumps in speedup as the variable values increases.

Using a QNX Trace

The tracelogger in QNX enables detailed tracing of the kernel and user process activity of QNX on any system. Thus, the logs or traces obtained from the tracelogger allows for a detailed inspection of the behaviour of the system. However, the developers and designers of a system running QNX have found it difficult to efficiently use such traces due to the fine-grained logging of events and the large length of the traces [30]. On the other hand, this drawback allows such traces to be an adequate resource for dynamic mining of system properties and patterns using TRE [30].

For this evaluation, we use an operational hexacopter running the QNX operating system to collect a trace containing 1.6 million events with 149 event types. We use the

Table 4.4: Mining $(\hat{}(P|S)^*.(\langle P.\hat{}(P|S)^*.S.\hat{}(P|S)^*\rangle[0,2000]))+$ on a QNX Trace

π_k	P	S	δ_{π_k}	ω_{π_k}
π_1	KER_EXIT-SIGNAL_ACTION/29	KER_CALL-SIGNAL_ACTION/29	0.883	4473
π_2	KER_CALL-SIGNAL_ACTION/29	KER_EXIT-SIGNAL_ACTION/29	0.458	2325
π_3	KER_CALL-SYS_CPUPAGE_GET/07	KER_EXIT-SYS_CPUPAGE_GET/07	0.388	104
π_4	INT_HANDLER_EXIT-0x00000044	INT_HANDLER_ENTR-0x00000044	0.373	13900
π_5	THREAD-THRUNNING	COMM-REC_MESSAGE	0.314	79327
π_6	KER_EXIT-SYNC_CREATE/78	KER_CALL-SYNC_CREATE/78	0.251	91
π_7	USREVENT-EVENT-2	KER_CALL-MSG_RECEIVEV/14	0.208	27300

alternating TRE template t_{TRE_A} to mine properties from the QNX trace. Table 4.4 shows the mining results with $\delta_{\pi_k} \geq 0.2$ along with the corresponding ω_{π_k} values sorted by δ_{π_k}.

From Table 4.4, alternating pattern represented by π_1 was mined successfully 4473 times, giving a confidence of around 0.88. However, the pattern with the values of P and S in π_1 swapped and represented by π_2 was mined successfully only 2325 times with a confidence of around 0.46. This can imply that the event types represented by P and Q in π_1 alternate relatively more often within the time interval of 2000 nanoseconds than π_2.We can also use a template of the form

$$(\langle P.(\hat{}(P|S|Q))^+.S.(\hat{}(P|S|Q))^+.Q\rangle[0,3000])+$$

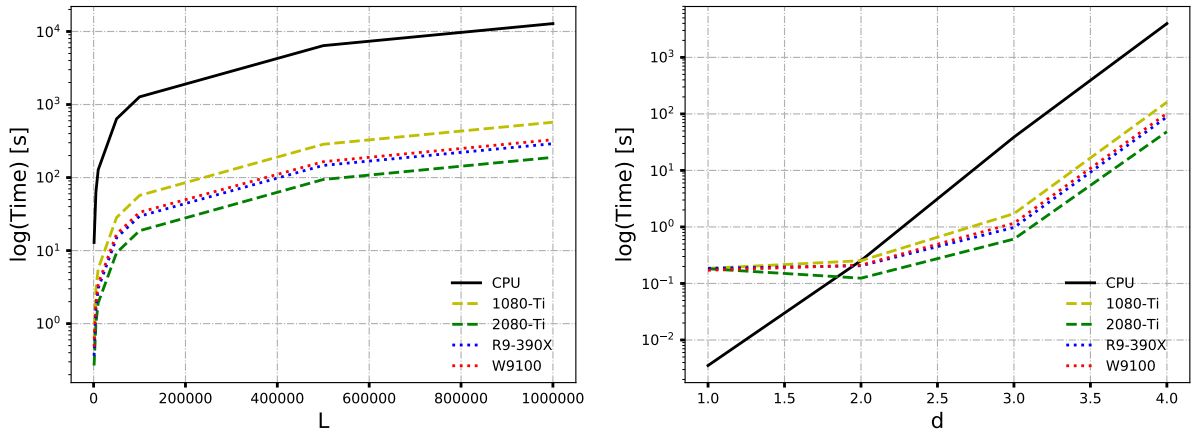
to mine patterns indicating the execution of any three event types with some interleaving between them within a time interval of 3000 nanoseconds. We noticed that the mining took around 2.5 hours on GeForce GTX 1080 Ti and the sequential version of this experiment was aborted as it exceeded the limit of 24 hours.

4.5.2 Case Study: Nested Words (NW)

Using Synthesized Traces on GPUs

The setups used for this case study are the following:

- **Setup 1** has varying L with $|\Sigma| = 152$, $d = 3$ and the results are shown in Figure 4.6a.
- **Setup 2** has varying d with $L = 10000$, $|\Sigma| = 98$ and the results are shown in Figure 4.6b.
- **Setup 3** has varying $|\Sigma|$ with $L = 500000$, $d = 3$ and the results are shown in Figure 4.6c.



(a) NW Mining Setup 1

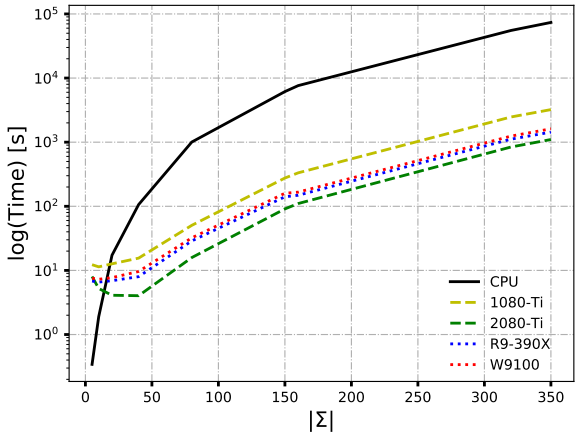
(b) NW Mining Setup 2

Figure 4.6: NW Mining Execution Time Evaluation on Synthesized Traces (Setups 1 and 2)

The NW template

$$[\langle a]^n.c + .[\rangle b]^n; n > 0$$

known as *Nested-Call-Processes-Response* pattern [29] is used to evaluate cases where d is constant (set to 3 in this case). This template is modified by adding or removing variables to evaluate for experiments where d is varied.



(c) NW Mining Setup 3

Figure 4.6: NW Mining Execution Time Evaluation on Synthesized Traces (Setup 3)

Table 4.5: NW Mining Speedup Results

Constant Parameters	Variable Parameter	Variable Values	1080 Ti Speedup	2080 Ti Speedup	R9-390X Speedup	W9100 Speedup
$d = 3$ $\|\Sigma\| = 152$	L	1000	21.569	48.329	35.443	26.736
		5000	21.880	63.719	40.325	34.614
		10000	22.525	66.432	41.936	36.386
		50000	22.488	68.075	42.480	37.807
		100000	22.371	68.056	43.069	38.093
		500000	22.416	68.051	43.623	38.826
		1000000	22.428	67.843	44.061	38.896
$L = 10000$ $\Sigma = 98$	d	1	0.019	0.019	0.019	0.021
		2	1.006	2.050	1.236	1.205
		3	22.625	63.646	39.730	33.416
		4	24.570	81.793	44.748	38.716
$d = 3$ $L = 500000$	$\|\Sigma\|$	5	0.028	0.043	0.050	0.045
		10	0.167	0.363	0.289	0.261
		20	1.362	4.213	2.497	2.238
		40	6.837	26.352	13.299	11.036
		80	19.917	63.383	34.868	31.194
		150	22.294	67.559	43.736	38.522
		160	22.931	69.073	51.285	45.838
		320	22.354	65.710	49.600	44.426
		350	22.879	67.003	51.351	45.585

The behaviour observed here is similar to that of the TRE mining experiments on synthesized traces. The plots demonstrate significant improvement in execution through the parallel implementations with Nvidia GeForce RTX 2080 Ti performing relatively better than the other devices and R9-390X providing the best performance among the OpenCL devices. d is set to 3 as the templates in NW mining are more expressive at $d \geq 3$. More the alphabet size and/or the number of template variables, more are the number of patterns to be mined which thereby, shows the significance of parallelism. Table 4.5 tabulates the speedup obtained from each GPU. Similar to TRE mining, consistent speedups are observed for setup 1 for each GPU. For setups 2 and 3, speedups increases as d and $|\Sigma|$ increases.

Using Twitter Feed

Nested words (NWs) are a model for the representation of data with both linear ordering and a hierarchical structure [2]. Examples of such data are XML documents and recursive program traces [2]. Nowadays, JavaScript code can be embedded freely inside HTML documents, and this feature can cause code injection attack called Cross-Site Scripting

Table 4.6: Mining $[\langle a]^n.[\langle c]^m.[\rangle d]^m.[\rangle b]^n$; $m, n > 0$ on a Twitter Feed

π_k	a	c	d	b	δ_π	ω_π
π_1	<svg>	<style>	</style>	<circle>	0.318	28
π_2	<style>	</style>	<svg>	<circle>	0.280	28
π_3	</button>			<button>	0.233	5766
π_4	<hr>	<fieldset>	<legend>	</legend>	0.167	1
π_5		<button>	</button>		0.135	4124
π_6			<button>	</button>	0.115	3302

(XSS) on web applications [21]. Such injections can transform the code into a vulnerable computer program, violate the nesting structure and change the course of execution. The specification mining technique with NW can mine properties with a specific pattern and these properties can then be used as HTML filters to verify secure web applications [29].

To demonstrate the results of mining NW using this framework, an HTML document representing a Twitter feed is used. This HTML document contains 241131 tags (events) in total with 76 unique tags (event types). Suppose, to mine properties that can showcase an equal number of opening and closing tags nested in between another equal number of opening and closing tags, the template

$$[\langle a]^m.[\langle c]^n.[\rangle d]^n.[\rangle b]^m; m, n > 0$$

known as the *Nested-Chained-Call-Response* pattern [29] can be used. The results of the mining are tabulated in Table 4.6. The table shows the mined patterns with $\delta_{\pi_k} \geq 0.1$ along with the corresponding ω_{π_k} values sorted by δ_{π_k}.

The mined results also contain singleton HTML tags (i.e. tags that don't require a closing HTML tag) as shown in the table. An example of such a pattern can be the pattern represented by π_1 where the singleton HTML tags are <svg> and <circle>. This pattern was mined successfully 28 times with a confidence of 0.318 with $m, n > 0$. A pattern with no singleton tags is π_5. This pattern was mined successfully 4124 times with a confidence of 0.135 with $m, n > 0$.

Chapter 5

Common Abstractions

This chapter discusses the common abstractions that were considered when programming the above algorithmic formulations for GPU computing. These abstractions are techniques that are common to both CUDA and OpenCL programming model and can be applied to explore improvements in GPU computing.

5.1 Grid/Global Size Strided Loops in a Kernel

Examples shown in Figure 2.4 and Figure 2.8 performs vector addition in parallel. However, these examples assume that the data array has size equal to the number of threads in the block or number of work-items in the work-group, i.e. 1024. This example does not apply when the size of the data array is less than or greater than the number of threads and work-items. For the case where the data array size is less, an *if* condition can be inserted before the addition operator. On the other hand, when the data array size is larger than the threads or work-items launched, a grid or global size strided *for* loop as shown in Figure 5.1 can be used to ensure all elements in a data array is covered.

By using such a loop on a large data array, the threads or work-items are reused to cover all the elements in the array. It also ensures the memory is accessed using memory coalescing pattern that is discussed in the next section. In addition to this, a device has a maximum number of threads or work-items and there can be cases where the sizes of data arrays in consideration can be larger than this maximum limit. These cases can be covered by using these loops in a kernel. This maximum number of threads or work-items also changes from device to device. Therefore, these strided loops enable portability between

```
1 __kernel void doSumStrided(global int *a, global int *b, global int *
      sum, int N) {
2     for (int index = get_global_id(0); index < N; index +=
          get_global_size(0)) {
3         sum[index] = a[index] + b[index];
4     }
5 }
```

Figure 5.1: Work-Group Strided Loop Example in OpenCL

different OpenCL-compliant devices or among the CUDA GPUs. Lastly, it also improves the readability of kernels when sequential *for* loops are parallelized using kernels.

5.2 Memory Coalescence

```
1 __kernel void doSumUncoalesced(global int *a, global int *b, global
      int *sum, int offset, int N) {
2     int i = get_global_id(0);
3     for (int index = i*offset; (index < ((i*offset) + offset)) &&
          index < N; ++index) {
4         sum[index] = a[index] + b[index];
5     }
6 }
```

Figure 5.2: Uncoalesced Memory Access Kernel Example in OpenCL

As mentioned in the discussion of the memory models, the global memory in the GPU is the largest memory available with high latency and is used when transferring large data from the CPU to the GPU. Memory coalescing is a memory access pattern to provide a way to access global memory efficiently. The load and store operations on global memory are done by coalescing threads or work-items together into wraps or wavefronts respectively. The highly recommended global memory access is when all the consecutive threads in a wrap or work-items in a wavefront access contiguous memory locations as the GPU accesses global memory data by retrieving a large chunk of the memory even if a subset of the chunk is to be used [9]. Hence, if consecutive threads or work-items access memory locations that are far apart, multiple chunks would have to be retrieved thereby impacting performance. This can be demonstrated using the vector addition example in OpenCL.

The kernel in Figure 5.2 shows uncoalesced memory access with each thread accessing a

set of contiguous memory locations. The `offset` parameter in this kernel is equal to the size of the data array (`N`) divided by the number of threads which in this case is 1024. A plot on the performance of this kernel against the coalesced memory access kernel in Figure 5.1 on an AMD FirePro W9100 GPU is shown in Figure 5.3. As expected, kernel with coalesced memory access, `doSumStrided` performs better than the kernel with uncoalesced memory access, `doSumUncoalesced`.

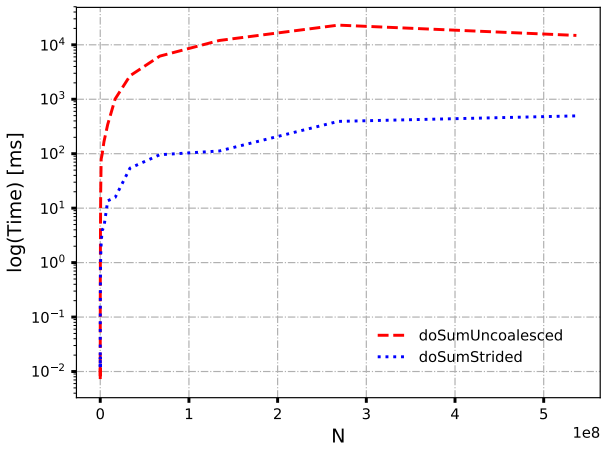


Figure 5.3: Time Comparison Between Coalesced and Uncoalesced Memory Access

5.3 Algorithmic Design for Parallelization

```
1 void update(vector<int> &in, vector<int> &out, int &n, int &a, int &b
      ) {
2     for (int i = 0; i < n; ++i) {
3         int temp = ((in[i] - b)/a);
4         out[temp] = max(out[temp], i);
5     }
6 }
```

Figure 5.4: Example Sequential *for* Loop with Dependencies

Identification of parallelism in a sequential algorithm is a key aspect of GPU computing. There can be cases where the efficient parallelism may not be evident for a given algorithmic formulation due to dependencies among other things. Hence, there is a need

```
1  __kernel void update(global int *in, global int *out, int n, int a,
       int b) {
2      for (int i = get_global_id(0); i < n; i+= get_global_size(0)) {
3          int temp = ((in[i] - b)/a);
4          atomic_max(&(out[temp]), i);
5      }
6  }
```

Figure 5.5: OpenCL Kernel v1 for the Sequential `update` Function

to restructure the algorithm to identify the parallelism involved without affecting the correctness effectively. Usually, the restructuring comes with an extra cost such as atomic operations or increasing space complexity. This can be demonstrated using a simple *for* loop example with dependencies. Figure 5.4 shows an example function with a *for* loop that calculates the index of the output array `out` and then stores the maximum index of the input array `in`. The assumptions in this example are that the input array `in` is sorted in ascending order and `b` is always less than any element in `in`.

```
1  __kernel void update1(global int *in, global int *temp, int n, int a,
       int b) {
2      for (int i = get_global_id(0); i < n; i += get_global_size(0)) {
3          temp[i] = ((in[i] - b)/a);
4      }
5  }
6
7  __kernel void update2(global int *temp, global int *out, int n) {
8      for (int i = get_global_id(0); i < n - 1; i+= get_global_size(0))
           {
9          if(temp[i] != temp[i+1]) {
10             out[temp[i]] = i;
11         }
12     }
13 }
```

Figure 5.6: OpenCL Kernel v2 for the Sequential `update` Function

Since the index to the output array is calculated inside the loop, there can be cases where that index happens to be the same for two or more iterations. This can result in a race condition if the loop is directly parallelized. Given this data dependency, one way to parallelize this loop is by applying the atomic operation on the `max` function using the `atomic_max` primitive in OpenCL as shown in Figure 5.5. Atomic operations can be executed by only one thread or work-item at a time thereby overcoming the data

dependency.

Another way to parallelize to increase the space complexity by splitting the function into two kernels such that the first kernel, `update1` executes line 3 of the update function and stores the results in a temporary array, `temp`. A second kernel, `update2` uses `temp` to finish the computation as shown in Figure 5.6. However, the second kernel is a bit different for parallelism purposes. Given that `in` is in increasing order, the values stored in `temp` will always be in increasing order as well and the maximum index is the index of the last occurrence of an element in `temp`.

Table 5.1: Time Measurements for Algorithmic Restructuring

N	Input	Average Time [s] OpenCL kernel v1	Average Time [s] OpenCL kernel v2
1×10^6	Evenly Increasing	0.0040	0.0055
1×10^6	Unevenly Increasing	0.0132	0.0148

The notion behind these different kernels is to see how different implementations of the same function perform for different inputs and help determine which one offers better time performance. The average time measurements over ten runs for each of the parallel implementations are shown in Table 5.1. The kernels were executed on a AMD R9-390X GPU. From the table, it can be seen that both kernels are quite close in performance but, the kernel with atomic operation provides slightly better timing measurement for both inputs in this specific example. Hence, there can be more than one way to parallelize a particular code segment if possible, and it is essential to evaluate the different parallelism possibilities to use the GPU effectively.

5.4 Determining the Number of Threads/Work-Items

In addition to experimenting with the structure of writing kernel code, it is also essential to determine the block-thread configuration or work-item configuration (or in general, kernel configuration) when launching or enqueueing a kernel. Recent GPUs have the facility to dispatch a large number of threads resulting in several possible configurations and hence, it is vital to determine the configurations to utilize the GPU effectively.

Intuition 1: More the number of blocks in CUDA or more the number of work-groups in OpenCL, better is the performance [11, 4].

As each block can be executed by a SM in CUDA and each work-group can be executed by a CU in OpenCL, it is important to set the number of blocks or work-groups to be at least the number of SMs or CUs. This prevents idle SMs or CUs. Unlike in CUDA where the number of thread blocks can be specified, OpenCL only provides a way to determine the total number of work-items (i.e. global work size G) and the number of work-items in each work-group (i.e. local work size L). OpenCL then calculates the number of work-groups which is $\frac{G}{L}$.

Intuition 2: The number of threads in a block to be a multiple of the number of threads in a wrap (32) and the local work size to be a multiple of the number of work-items in a wavefront (64) [11, 4].

This is because each wrap or each wavefront is the fundamental unit of work in CUDA or OpenCL GPU respectively. The performance can be affected negatively if this intuition is not followed. The OpenCL vector addition example shown in Figure 5.1 can be used to demonstrate the impact of kernel configuration on the performance. In addition to these intuitions, there are performance counters that affect the configurations, and these can be evaluated using profilers such as nvprof for CUDA and CodeXL for OpenCL in AMD GPUs. Lastly, it is also important to keep in mind that each device has a limit on the number of threads in a block or the number of work-items in a work-group.

Table 5.2: Time Measurements for Different OpenCL Kernel Configurations

Configuration	Global Work Size	Local Work Size	Time [ms]
C1	(1024, 1, 1)	(64, 1, 1)	482.533
C2	(N, 1, 1)	NULL	45.067
C3	(N/2, 1, 1)	NULL	50.31896
C4	(N, 1, 1)	(32, 1, 1)	80.052
C5	(N, 1, 1)	(64, 1, 1)	47.213
C6	(N, 1, 1)	(65, 1, 1)	64.283

The results for using different configurations of the kernel in Figure 5.1 on an AMD R9-390X GPU are tabulated in Table 5.2. N in the table is the size of the input arrays, and it is set to $536,870,912$ ($512 \times 1024 \times 1024$) elements. The following observations can be made:

- C1 takes the longest as it has the lowest global work size of 1024 resulting in a small number of work-groups.

- C2 and C3 set the global work size to N and $\frac{N}{2}$ respectively with the local group size to `NULL`. This means that OpenCL will determine the local work size and in this case, CodeXL shows that OpenCL uses the maximum available local work size for the device, i.e. 256. C2 performs better, and this is by Intuition 1.

- However, C4 uses the most number of groups but performs slower than C2. This can be due to Intuition 2. This is verified by CodeXL using the `VALUUtilization` performance counter which is the percentage of active vALU threads in a wavefront. `VALUUtilization` for C4 is at 50% and at 100% for C2. According to the CodeXL documentation, the ideal value is 100%, and a lower value can be due to the work-group size not being a multiple of 64.

- Though C5 obeys Intuition 1 and 2, C2 still performs slightly better relatively. All performance counters except `FetchSize` and `WriteSize` are the same or very similar between the two configurations. `FetchSize` is the total kilobytes fetched from the memory and `WriteSize` is the total kilobytes written to the memory. C2 uses relatively lower values of `FetchSize` and `WriteSize` which can be a possible explanation for better performance.

- C6 is another example of not setting local work size to be a multiple of 64. An increase by one work-item on the local work size of C5 results in an increase by around 20 ms in time. This can be due to another wavefront being launched for that one extra work-item, and though the remaining 63 work-items in the wavefront are inactive, they still consume the resources in a CU [11].

Similar observations can be made in Nvidia GPUs using the CUDA profiler, nvprof which offers several different performance counters such as `Global Load Throughput` and `Global Memory Load Efficiency` [11].

5.5 Vector Datatypes

In addition to the scalar data types such as `int`, `float`, OpenCL and CUDA also provide vector data types which are defined with the scalar type name followed by a suffix n such as `int2`, `int4`, `float2` and `float4`. The value of n indicates the number of elements that can be held by the vector and the supported values are $\{2, 3, 4, 8, 16\}$. The detailed specifications on the supported vector data types can be found in the OpenCL and CUDA documentations. The elements in the vector can be accessed using `.x`, `.y`, `.z` and `.w` notations as shown in the following example:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float a = f.x; // a = 1.0
float b = f.y; // b = 2.0
float c = f.z; // c = 3.0
float d = f.w; // d = 4.0
```

One benefit of using a variable declared as a vector data type is that the elements in the vector are operated on at the same time by a work-item and thread [39]. This is done by packing multiple instructions into a Very Long Instruction Word (VLIW) thereby reducing the total number of instructions. In addition to this, vector data types also provide better timing performance in specific cases, and this can be demonstrated using an example. The kernel `add1` in Figure 5.7 shows a vector addition example using the `float` data type.

```
1 __kernel void add1(global float *a, global float *b, global float *
      out, long N) {
2     for (long i = get_global_id(0); i < n; i+= get_global_size(0)) {
3         out[i] = a[i] + b[i];
4     }
5 }
```

Figure 5.7: Vector Addition Kernel Using `float` Scalar Data Type

```
1 __kernel void add2(global float2 *a, global float2 *b, global float2
      *out, long n) {
2     for (long i = get_global_id(0); i < n; i+= get_global_size(0)) {
3         out[i] = a[i] + b[i];
4     }
5 }
```

Figure 5.8: Vector Addition Kernel Using `float2` Vector Data Type

```
1 __kernel void add4(global float4 *a, global float4 *b, global float4
      *out, long n) {
2     for (long i = get_global_id(0); i < n; i+= get_global_size(0)) {
3         out[i] = a[i] + b[i];
4     }
5 }
```

Figure 5.9: Vector Addition Kernel Using `float4` Vector Data Type

This kernel is compared against two other kernels, `add2` and `add4` as shown in Figure 5.8 and Figure 5.9 respectively. The kernel `add2` uses the `float2` data type and the value

of n in this kernel is half of the number of elements processed by `add1`. Similarly, the kernel `add4` uses the `float4` data type and the value of n in this kernel is quarter of the number of elements processed by `add1`. Figure 5.10 shows a plot that illustrates the timing performance comparison between these kernels on an AMD R9-390X GPU. From the plot, it can be seen that all the kernels have similar performance in the beginning but as the value of N increases, `add4` gives the best performance at around $N = 5 \times 10^7$ and `add2` provides the best performance for $N > 1.3 \times 10^8$ in this setup.

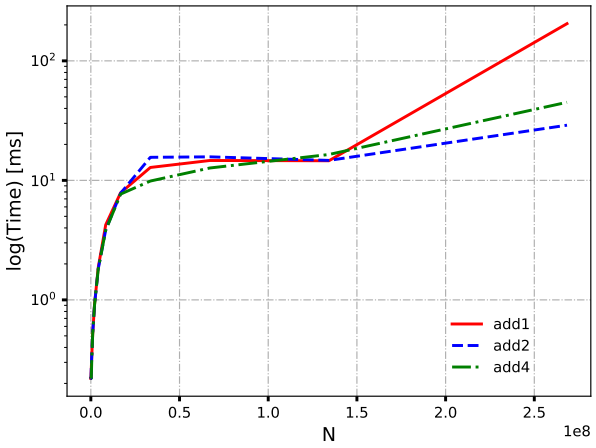


Figure 5.10: Performance Comparison Between a Scalar Data Type and Vector Data Types

Therefore, there are specific cases where vector data types have the potential to provide better performance than scalar data types. However, one caveat to vector data types is that the device memory pointer of a vector data type should be aligned to the size of the data type used and the size of the data type should be a power of two in bytes. Also, as the number of elements processed by `add2` and `add4` is less than that of `add1`, the overall parallelism has been reduced. Lastly, using vector data types increases the use of registers thereby, possibly impeding the performance of register-limited kernels. Therefore, vector data types are a potential catalyst for optimizations and should be used whenever possible.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we present an exploratory study on the acceleration of computationally demanding data analysis tools using parallel computing in heterogeneous systems. Applying parallel computing via heterogeneous systems are increasingly becoming valuable alternatives to overcome the performance limits of CPUs. This is demonstrated using two recently proposed computationally intensive data analysis tools - construction of arrival curves and regular specification mining - on a heterogeneous platform with CPU as the host and GPU as the device. For each such tool, we discuss the background, the algorithmic formulations to transform the problems into parallel computations and also, experiments using CUDA and OpenCL programming models to demonstrate the acceleration obtained. The experiments showed that heterogeneous platforms with a GPU outperform equivalent sequential implementations with significant speedups. Thus, to summarize the major contributions of this thesis:

- Presentation of novel approaches to construct empirical arrival curves and to mine specifications using the regular language class of automaton (regular specification mining) from execution traces.
- Algorithmic breakdown of arrival curve construction and regular specification mining to expose the high-degree of data-level parallelism involved and showcase the significant accelerations obtained using GPUs.
- Optimizations that can be considered when applying parallel computing to computationally intensive tasks using GPUs.

6.2 Future Work

The parallel approaches in both algorithms presented in this thesis are platform independent. This enables the application of the algorithms on other parallel processing technologies such as Open Multi-Processing (OpenMP) or OpenCL in other devices. Specifically, as OpenCL allows portability by supporting devices with different architectures, future work can be to accelerate these algorithms in another heterogeneous system with a different OpenCL compliant device such as a Xilinx FPGA. Xilinx, a manufacturer of FPGA, also collaborates with the Khronos Group to develop and support OpenCL applications for Xilinx FPGA devices using the Xilinx SDAccel development environment.

Preliminary experiments conducted by porting with minimal or no changes of the TRE mining developed for AMD GPUs onto a Xilinx Kintex UltraScale FPGA KCU1500 on the CMC Cloud Services did not result in speedups. The initial run with $L = 500000, |\Sigma| = 80, d = 2$ and no optimizations finished execution after 4 hours. This showed that direct porting of OpenCL code from an AMD GPU to a Xilinx FPGA device cannot guarantee acceleration. Two key observations in OpenCL for a Xilinx FPGA were the limited number of work-items and the strided loops discussed in Section 5.1 being ineffective. This resulted in assigning a range of contiguous memory locations to each of the limited set of work-items as the first optimization. In addition to this, loop unrolling inside the kernels brought the execution time down to 40 minutes. Likewise, other optimizations such as using multiple Double Data Rate (DDR) banks, using multiple command queues and many more as recommended in the SDAccel Optimization Guide need to be applied to use the Xilinx FPGA for acceleration efficiently. Also, the long compilation process in current OpenCL tools for FPGA should be considered during development.

Unlike a GPU device where the architectural design allows developers to use off-the-shelf hardware readily, a FPGA device starts as an empty computational canvas and allows the developers to customize the device to fit an application or a class of applications best. Additionally, as discussed in recent literature, FPGAs are becoming more valuable nowadays due to relatively better performance-to-power efficiency when compared to GPUs, and this can serve as a good motivation for future work.

References

[1] Rajeev Alur and Dana Fisman. Colored Nested Words. In Adrian-Horia Dediu, Jan Janoušek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 143–155, Cham, 2016. Springer International Publishing.

[2] Rajeev Alur and P. Madhusudan. Adding Nesting Structure to Words. *J. ACM*, 56(3):16:1–16:43, May 2009.

[3] AMD. Whitepaper - AMD Graphics Core Next (GCN) Architecture. Technical report, AMD, Jun 2012.

[4] AMD. AMD APP SDK - OpenCL Optimization Guide. Technical report, AMD, Aug 2015.

[5] AMD. AMD Graphics Core Next Architecture, Generation 3. Technical report, AMD, Aug 2016.

[6] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A Distributed, Scalable Platform for Data Mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[7] G. Carvajal, M. Salem, N. Benann, and S. Fischmeister. Enabling Rapid Construction of Arrival Curves From Execution Traces. *IEEE Design Test*, 35(4):23–30, Aug 2018.

[8] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010.

[9] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2014.

[10] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 5–14, New York, NY, USA, 2008. ACM.

[11] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[12] Maxime Crochemore and Christophe Hancart. *Automata for Matching Patterns*, pages 399–462. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[13] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating Test Cases for Specification Mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[14] Azadeh Farzan, Matthias Heizmann, Jochen Hoenicke, Zachary Kincaid, and Andreas Podelski. Automated Program Verification. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 25–46, Cham, 2015. Springer International Publishing.

[15] Mark Gabel and Zhendong Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[16] Mark Gabel and Zhendong Su. Online Inference and Enforcement of Temporal Properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[17] Stanford VLSI Group. CPU DB - Looking At 40 Years of Processor Improvements - A complete database of processors for researchers and hobbyists alike. http://cpudb.stanford.edu/. Accessed: 2018-10-09.

[18] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[19] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[20] Kai Huang, Gang Chen, Christian Buckl, and Alois Knoll. Conforming the runtime inputs for hard real-time embedded systems. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, pages 430–436. ACM, June 2012.

[21] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 66–77, New York, NY, USA, 2014. ACM.

[22] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 178–189, New York, NY, USA, 2014. ACM.

[23] K. Lampka, S. Bondorf, J. B. Schmitt, N. Guan, and Yi W. Generalized finitary Real-Time calculus. In *Proceedings of IEEE INFOCOM*, 2017.

[24] Kai Lampka, Björn Forsberg, and Vasileios Spiliopoulos. Keep it cool and in time: With runtime monitoring to thermal-aware execution speeds for deadline constrained systems. *Journal of Parallel and Distributed Computing*, 95:79–91, 2016.

[25] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL Specification Mining. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92. IEEE, 2015.

[26] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable Specification Mining for Verification and Diagnosis. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 755–760, New York, NY, USA, 2010. ACM.

[27] David Lo and Siau-Cheng Khoo. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 265–275, New York, NY, USA, 2006. ACM.

[28] L. Mariani, M. Pezz, and M. Santoro. GK-Tail+ An Efficient Approach to Learn Software Models. *IEEE Transactions on Software Engineering*, 43(8):715–738, Aug 2017.

[29] Apurva Narayan, Nirmal Benann, and Sebastian Fischmeister. Mining Specifications using Nested Words. In *Proceedings of the 6th International Workshop on Software Mining*, Urbana-Champaign, USA, 2017.

[30] Apurva Narayan, Greta Cutulenco, Yogi Joshi, and Sebastian Fischmeister. Mining Timed Regular Specifications from System Traces. *ACM Trans. Embed. Comput. Syst.*, 17(2):46:1–46:21, January 2018.

[31] Colm Networks. Ragel State Machine Compiler. http://www.colm.net/open-source/ragel/. Accessed: 2017-07-26.

[32] M. Neukirchner, P. Axer, T. Michaels, and R. Ernst. Monitoring of Workload Arrival Functions for Mixed-Criticality Systems. In *IEEE 34th Real-Time Systems Symposium (RTSS)*, pages 88–96, Dec 2013.

[33] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu chun Feng, and Michela Becchi. Demystifying automata processing: GPUs, FPGAs or Micron's AP? In *ICS*, 2017.

[34] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[35] Michael Pradel and Thomas R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Vahid Rahmanzadeh and Mohammad Bagher Ghaznavi-Ghoushchi. *A Multi-Gb/s Parallel String Matching Engine for Intrusion Detection Systems*, pages 847–851. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[37] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach*. PhD thesis, Technical University Carolo-Wilhelmina of Braunschweig, 2005.

[38] Mahmoud Salem, Mark Crowley, and Sebastian Fischmeister. Anomaly Detection Using Inter-Arrival Curves for Real-Time Systems. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–106, 2016.

[39] Matthew Scarpino. *OpenCL in action: how to accelerate graphics and computation*. Manning Pub., Shelter Island, NY, 2011.

[40] Eric Schadt, Michael D Linderman, Jon Sorenson, Lawrence Lee, and Garry Nolan. Computational solutions to large-scale data management and analysis. *Nature reviews Genetics*, 11:647–57, 09 2010.

[41] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static Specification Mining Using Automata-Based Abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, Sept 2008.

[42] Hauke Stähle. *A Model-Based Framework for System-Wide Plug-and-Play with Flexible Timing Verification for Automotive Systems*. PhD thesis, München, Technische Universität München, Diss., 2016, 2016.

[43] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[44] S Sumathi and S.N. Sivanandam. *Introduction to Data Mining Principles*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[45] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[46] Dogan Ulus. Montre: A Tool for Monitoring Timed Regular Expressions. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 329–335, Cham, 2017. Springer International Publishing.

[47] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[48] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, 2006.

[49] Martin H. Weik. *pattern-matching*, pages 1240–1240. Springer US, Boston, MA, 2001.

[50] Ren Wu, Bin Zhang, and Meichun Hsu. GPU-Accelerated Large Scale Analytics. Technical report, HP Labs, 2009.

[51] Xiaodong Yu and Michela Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 18:1–18:10, New York, NY, USA, 2013. ACM.